AFRL-RI-RS-TR-2017-033

STATIC PROGRAM ANALYSIS FOR RELIABLE, TRUSTED APPS

UNIVERSITY OF WASHINGTON

FEBRUARY 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-033 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEBRUARY 2017	FINAL TECHNICAL REPORT	FEB 2012 – SEP 2016

4. TITLE AND SUBTITLE

STATIC PROGRAM ANALYSIS FOR RELIABLE, TRUSTED APPS

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-12-2-0107

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Michael Ernst

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
97

5f. WORK UNIT NUMBER
73

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Washington
185 Stevens Way;Paul Allen Center for CSE
Seattle, WA 98195

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-033

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report presents three main contributions of the SPARTA project, which was part of DARPA's Automated Program Analysis for Cybersecurity (APAC) program. The first contribution, is a model for collaborative verification of information flow for a high-integrity app store. The second contribution is analyses for implicit control flow that improve precision of downstream analyses. The third contribution is improvements to the Checker Framework. The Checker Framework is an open-source tool that enhances Java's type system to make it more powerful and useful.

15. SUBJECT TERMS
Information Flow Analysis of Apps, Implicit Control Flow, Checker Framework

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK K. WILLIAMS**
U	U	U	UU	67	19b. TELEPHONE NUMBER (*Include area code*) **NA**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 SUMMARY

This report presents three main contributions of the SPARTA project, which was part of DARPA's APAC program.

The *first contribution*, is a model for collaborative verification of information flow for a high-integrity app store. Current app stores distribute some malware to unsuspecting users, even though the app approval process may be costly and time-consuming. High-integrity app stores must provide stronger guarantees that their apps are not malicious. We propose a verification model for use in such app stores to guarantee that the apps are free of malicious information flows. In our model, the software vendor and the app store auditor collaborate — each does tasks that are easy for her/him, reducing overall verification cost. The software vendor provides a behavioral specification of information flow (at a finer granularity than used by current app stores) and source code annotated with information-flow type qualifiers. A flow-sensitive, context-sensitive information-flow type system checks the information flow type qualifiers in the source code and proves that only information flows in the specification can occur at run time. The app store auditor uses the vendor-provided source code to manually verify declassifications.

We have implemented the information-flow type system for Android apps written in Java, and we evaluated both its effectiveness at detecting information-flow violations and its usability in practice. In an adversarial Red Team evaluation, we analyzed 72 apps (576,000 LOC) for malware. The 57 Trojans among these had been written specifically to defeat a malware analysis such as ours.

Nonetheless, our information-flow type system was effective: it detected 96% of malware whose malicious behavior was related to information flow and 82% of all malware. In addition to the adversarial evaluation, we evaluated the practicality of using the collaborative model. The programmer annotation burden is low: 6 annotations per 100 LOC. Every sound analysis requires a human to review potential false alarms, and in our experiments, this took 30 minutes per 1,000 LOC for an auditor unfamiliar with the app.

The manual for the SPARTA toolset, which includes the information-flow type checker and also other tools, is available https://types.cs.washington.edu/sparta/current/sparta-manual.pdf.

The *second contribution* is analyses for implicit control flow that improve precision of downstream analyses. Implicit or indirect control flow is a transfer of control between procedures using some mechanism other than an explicit procedure call. Implicit control flow is a staple design pattern that adds flexibility to system design. However, it is challenging for a static analysis to compute or verify properties about a system that uses implicit control flow.

We present static analyses for two types of implicit control flow that frequently appear in Android apps: Java reflection and Android intents. Our analyses help to resolve where control flows and what data is passed. This information improves the precision of downstream analyses, which no longer need to make conservative assumptions about implicit control flow.

We have implemented our techniques for Java. We enhanced an existing security analysis with a more precise treatment of reflection and intents. In a case study involving ten real-world Android apps that use both intents and reflection, the precision of the security analysis was increased on average by two orders of magnitude. The precision of two other downstream analyses was also improved.

The *third contribution* is improvements to the Checker Framework. The Checker Framework is an open-source tool that enhances Java's type system to make it more powerful and useful. This lets software developers detect and prevent errors in their Java programs. The Checker Framework includes compiler plug-ins ("checkers") that find bugs or verify their absence. It also permits you to write your own compiler plug-ins. Each plug-in replaces Java's type system with a more powerful one that prevents certain errors.

All of the work described so far was built upon the Checker Framework. Our work pushed the limits of what can be expressed as a pluggable type system. During the course of this work, we discovered and fixed bugs in the Checker Framework, but more importantly we enhanced its capabilities.

2 INTRODUCTION

2.1 COLLABORATIVE VERIFICATION OF INFORMATION FLOW

App stores make it easy for users to download and run applications on their personal devices. App stores also provide a tempting vector for an attacker. An attacker can take advantage of bugdoors (software defects that permit undesired functionality) or can insert malicious Trojan behavior into an application and upload the application to the app store.

For current app stores, the software vendor typically uploads a compiled binary application. The app store then analyzes the binary to detect Trojan behavior or other violations of the app store's terms of service. Finally, the app store approves and publishes the app. Unfortunately, the process offers few guarantees, and every major app store has approved Trojans [12,36,41,44,46,70,73,85].

We explored the practicality of a high-assurance app store that gives greater understanding of, and confidence in, its apps' behavior in order to reduce the likelihood that a Trojan is approved and distributed to users. A high-assurance app store would be particularly valuable in certain sensitive settings. For example, corporations already provide lists of apps approved for use by employees (often vetted by ad hoc processes). More relevantly to our sponsor, the U.S. Department of Defense is also actively pursuing the creation of high-assurance app stores.

Four contributing factors in the approval of Trojans by existing app stores are: (1) Existing analysis tools are poorly automated and hard to use; much manual, error-prone human effort is required. (2) The vendor provides only a very coarse description of application behavior in the form of permissions it will access: system resources such as the camera, microphone, network, and address book. This characterization provides insufficient limitations on the application's behavior. (3) The binary executable lacks much semantic information that is available in the source code but has been lost or obfuscated by the process of compilation. (4) The vendor has little incentive to make the application easy for the app store to analyze and understand.

We have developed a new approach to verifying apps that addresses each of these factors. (1) We have created a powerful, flow-sensitive, context-sensitive type system that verifies information flows. The type system is easy to use and works with Java and Android. (2) Our type system proves that apps conform to finer-grained information-flow specifications than current app stores. These specifications indicate not just which resources may be accessed but which information flows are legal — how the resources may be used by the program. (3) Our approach uses source code rather than binaries, because source code provides more information, enables more accurate and powerful analyses, and allows an auditor to evaluate false positive warnings. While not all application developers may wish to provide their source code to an app store, we argue that this requirement is reasonable for app stores in certain settings, e.g., in the context of corporate, military, government, or medical applications. (4) We propose a collaborative verification methodology in which the vendor participates in and contributes to the verification process, rather than casting the vendor and the app store in an antagonistic relationship. However, the developer is not trusted: all information provided by the developer is verified.

We report on initial experience with this system, including an adversarial Red Team exercise in which 5 corporate teams (funded externally, not by us) were given access to our source code and design documents then tasked with creating Trojans that would be difficult to detect. Our type system detected 82% of the Trojans, and 96% of the Trojans whose malicious behavior was related to information flow.

(We have identified an enhancement to our system that would increase the latter number to 100%.) As with any program analysis, a human must investigate tool warnings to determine whether they are false positives. On average, it took an auditor unfamiliar with the programs 30 minutes per KLOC to analyze the information flow policy and the tool warnings. The annotation burden for programmers (application vendors) is also low.

Overall, our goal is to make it difficult to write Trojans and easy to determine when code is not a Trojan. Our information-flow type-checker cannot catch all malware, but it raises the bar for malware authors and thus improves security.

2.2 STATIC ANALYSIS OF IMPLICIT CONTROL FLOW

Programs are easier to understand and analyze when they use explicit control flow: that is, each procedure call invokes just one target procedure. However, explicit control flow is insufficiently flexible for many important domains, so implicit control flow is a common programming paradigm. For example, in object-oriented dispatch a method call invokes one of multiple implementations at run time. Another common use of implicit control flow is in design patterns, many of which add a level of indirection in order to increase expressiveness. This indirection often makes the target of a procedure call more difficult to determine statically.

Implicit control flow is a challenge for program analysis. When a static analysis encounters a procedure call, the analysis usually approximates the call's behavior by a summary, which conservatively generalizes the effects of any target of the call. If there is only one possible target (as with a normal procedure call) or a small number that share a common specification (as with object-oriented dispatch), the summary can be relatively precise. But if the set of possible targets is large, then a conservative static analysis must use a very weak specification, causing it to yield an imprecise result.

The imprecision is caused by a lack of information about possible call targets and about the types of data passed as arguments at each call. Our goal is to provide a sound and sufficiently precise estimate of potential call targets and of the encapsulated data communicated in implicit invocations, in order to improve the precision of downstream program analyses.

Our evaluation focuses on a particular domain — Android mobile apps — in which implicit invocation significantly degrades static analysis. In our experience [23], the largest challenge to analyzing Android apps is their use of reflection and intents, and this led us to our research on resolving implicit invocation. We are not aware of a previous solution that handles reflection and intents soundly and with high precision.

Reflection permits a program to examine and modify its own data or behavior [74]. Our interest is in use of reflection to invoke procedures. For example, in Java an object *m* of type Method represents a method in the running program; *m* can be constructed in a variety of ways, including by name lookup from arbitrary strings. Then, the Java program can call `m.invoke(...)` to invoke the method that *m* represents. Other programming languages provide similar functionality, including C#, Go, Haskell, JavaScript, ML, Objective-C, PHP, Perl, Python, R, Ruby, and Scala.

Android intents are the standard inter-component communication mechanism in Android. They are used for communication within an app (an app may be made up of dozens of components), between apps, and with the Android system. An Android component can send or broadcast intents and can register interest in receiving intents. The Android architecture shares similarities with blackboard systems and other message-passing and distributed systems.

By default, a sound program analysis must treat reflection and intents conservatively — the analysis must assume that anything could happen at uses of reflection and intents, making its results imprecise. We have built a simple, conservative, and quite precise static analysis that models the effects of reflection and intents on program behavior. The key idea is to resolve implicit control and data flow first to improve the estimates of what procedures are being called and what data is being passed; as a result, those constructs introduce no more imprecision into a downstream analysis than a regular procedure call does.[1]

Both control flow and data flow are important. For reflection, our approach handles control flow by analyzing reflective calls to methods and constructors to estimate which classes and methods may be manipulated, and it handles data flow via an enhanced constant propagation. For intents, our approach handles control flow by using previous work [64] to obtain component communication patterns, and it handles data flow by analyzing the payloads that are carried by intents.

We have implemented our approach for Java. We evaluated our implementation on open-source apps, in the context of three existing analyses, most notably an information flow type system for Android security [23]. Most Android apps use reflection and/or intents, so accurately handling reflection and intents is critical in this domain. Unsoundness is unacceptable because it would lead to security holes, and poor precision would make the technique unusable due to excessive false-positive alarms. The reflection and intent analyses increased the precision of the information flow type system

[1] Our approach does not change the program's operations, either on disk or in memory in the compiler.

by two orders of magnitude, and they also improved the precision of the other two analyses. Furthermore, they are easy to use and fast to run. Our implementation is freely available in the SPARTA toolset (`http://types.cs.washington.edu/sparta/`), including source code and user manual, and the reflection analysis is also integrated into the Checker Framework (`http://checkerframework.org/`).

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 COLLABORATIVE VERIFICATION OF INFORMATION FLOW

3.1.1 Introduction

An app store can be made more secure by requiring vendors to provide their applications in source code, and then performing strong verification on that source code. While today's commercial app stores do not require source code, we discuss in 3.1.1.1 the market forces that enable an app store such as we propose. This app store would analyze the source code, compile it, and distribute it as a binary (signed by the app store's private key) to protect the vendor's intellectual property. Availability of source code fundamentally changes the approval process in favor of verification by providing more information to both the analysis and the analyst.

Source code verification is relevant for other domains than high-integrity application stores. One public example of inserting malicious behavior into an open source program is an attempt to insert a backdoor in the Linux kernel [45]. As another example, Liu et al. developed proof-of-concept malware as Chrome extensions [53], which are essentially distributed as source code. The Heartbleed bug appeared in open-source software. We believe that source code analysis for security will become increasingly important, so it is worthy of attention from security researchers.

Our approach is for Java source code, but since the type qualifiers are persisted to the classfile, it would be possible to re-implement our type system for bytecode in order to verify compiled apps.

3.1.1.1 Collaborative verification model

Most app store approval policies assume an adversarial, or at least non-cooperative, relationship between the developer and the app store. The developer delivers an app in binary form, and the app store uses an opaque process to make a decision about whether to offer the app on the app store.

We propose to augment existing app store approval processes with a collaborative model (**Figure 1**) for verification of information flow. The application vendor provides more information to the auditor (an app store employee). This information is easy for the vendor to provide, but it would be difficult for the auditor to infer. The auditor is able to make a decision about information flow more quickly and with greater confidence, which is advantageous to both parties.

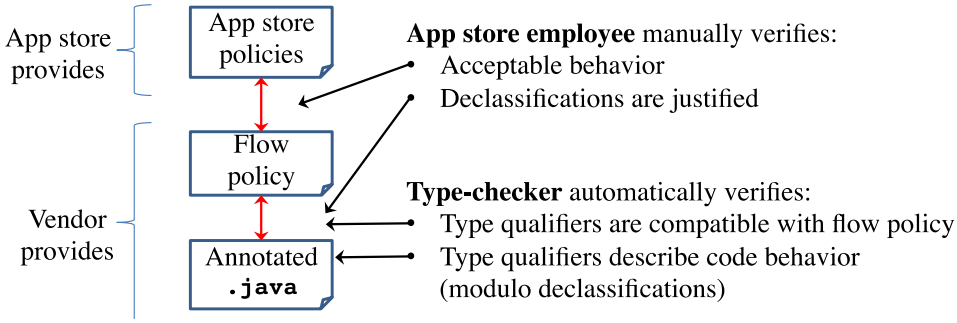

Figure 1 The collaborative verification model for information flow. The flow policy is a high-level specification that expresses application behavior in terms of user-visible information flows.

As shown in **Figure 1**, the auditor receives two artifacts from the vendor. The first vendor-provided artifact is the flow policy, a high-level specification of the intended information flows in the program from the user point of view. In our experiments, this averaged 6 lines long. For example, it might state that location information is permitted to flow to the network and that camera images may be written to the local disk. Any information flow not stated in the flow policy is implicitly forbidden. The second vendor-provided artifact is the source code, annotated with information flow type qualifiers. The annotation burden is low: on average 6 annotations per 100 lines of code.

Both the annotations and the vendor are untrusted. Our implementation, Information Flow Typechecker (IFT), automatically ensures that the type qualifiers are both permitted by the flow policy and are an accurate description of the source code's behavior (modulo any auditor-verified declassifications). If not, the app is rejected. Unannotated apps are also rejected. Thus, the application vendor must provide accurate type qualifiers and flow policy.

The auditor has two tasks, corresponding to the two vendor-provided artifacts. The first task is to evaluate the app's flow policy. This is a manual step, in which the auditor compares the flow policy to the app's documentation and to any app store or enterprise policies. The app store analyst must approve that the requested flows are reasonable given the app's purpose; apps with unreasonable flow policies are rejected as potential Trojans. The second task is to verify each declassification, using some other verification methodology (e.g., [9]). Sect. 4.1.3.1 further describes the auditing process.

Not every app store will desire to differentiate itself through increased security, and not every vendor will desire to participate in high-assurance app stores. But market forces will enable such stores to exist where there are appropriate economic incentives — that is, whenever some organizations or individuals are willing to pay more for increased security. Increased security is especially important in sensitive contexts such as government, corporate, and medical applications. Even if some vendors will never participate in a high-assurance app store, we believe there is value in researchers investigating and improving the practicality of such stores.

It makes economic sense for the vendor to annotate their code and possibly to be paid a premium: based on our experience, the effort is much less for the author of the code than for an auditor who would have to reverse-engineer the code before writing

down the information about the information flows. The effort is small compared to overall development time and is comparable to writing types in a Java program. If the type qualifiers are written as the code is first developed, they may even save time by preventing errors or directing the author to a better design.

Some vendors may be concerned with confidentiality of their source code. Large organizations already require their vendors to provide and/or escrow source code. For Android apps, it is easy to decompile a Java program from .class or .dex format, so even an app in binary format does not protect the vendor's algorithms, protocols, and other secrets. These facts may reduce vendors' reluctance to provide source code.

The U.S. Department of Defense is also interested in high-assurance app stores, for example through DARPA's "Transformative Apps" and "Automated Program Analysis for Cybersecurity," along with related software verification programs such as "High-Assurance Cyber Military Systems" and "Crowd-Sourced Formal Verification". Although it is funded by APAC, our collaborative verification model is novel and differs from the proposals on which DARPA's programs were built.

3.1.1.2 Threat model

While there are many different types of malicious activities, we focus on Trojans whose undesired behavior involves information flow from sensitive sources to sensitive sinks. This approach is surprisingly general: we have found that our approach can be adapted to other threats, such as detecting when data is not properly encrypted, by treating encryption as another type of resource or permission.

More specifically, IFT uses a flow policy as a specification or formal model of behavior. If IFT issues no warnings, then the app does not permit information flows beyond those in the flow policy

— that is, each output value is affected only by inputs specified in the flow policy. IFT issues a warning at every declassification, and manual checking is required for each one. IFT does not perform labor-intensive full functional verification, only information-flow verification, which we show can be done at low cost.

Our threat model includes the exfiltration of personal or sensitive information and contacting premium services. However, it does not cover phishing, denial of service, or side channels such as battery drain or timing. It does not address arbitrary malware (such as Slammer, Code Red, etc.). We treat the operating system, our type checker, and annotations on unverified libraries as trusted components — if they have vulnerabilities or errors, then an app could be compromised even if it passes our type system. App developers and app source code (including type qualifiers) are not trusted. There have been previous studies of the kinds of malware present in the wild [28, 93]. Felt et al. [28] classify malware into 7 distinct categories based on behavior. Our system can catch malware from the 4 most prevalent and important ones: stealing user information (60%), premium calls or SMSs (53%), sending SMS advertising spam (18%), and exfiltrating user credentials (9%). The other 3 categories are: novelty and amusement (13%), search engine optimization (2%), ransom (2%).

Our approach is intended to be augmented by complementary research and app store activities that focus on other threats. Our approach raises the bar for attackers rather than providing a silver bullet.

Sect. 3.2.8.1discusses limitations of our system in greater detail.

3.1.1.3 Contributions

The idea of verifying information flow is not new, nor is using a type system. Rather, our contributions are a new design that makes this approach practical for the first time, and realistic experiments that show its effectiveness. In particular, the contributions are:

We have proposed a collaborative verification model that reduces cost and uncertainty, and increases security, when investigating the information flow of apps submitted to an app store. Our work explores a promising point in the trade-off between human and machine effort.

We have extended information-flow verification to a real, unmodified language (Java) and platform (Android). Our design is easy to use yet supports polymorphism, reflection, intents, defaulting, library annotations, and other mechanisms that increase expressiveness and reduce human effort.

We have designed a mechanism for expressing information flow policies, and we have refined the existing Android permission system to make it less porous.

We have implemented our design in a publicly-available system (`http://types.cs.washington.edu/sparta/`), and we have experimentally evaluated our work. Our system effectively detected realistic malware targeted against it, built by skilled Red Teams. The effort to use our system was low for both programmers and auditors: our system is powerful, yet it requires less annotation overhead than previous systems and is simpler to use and understand.

3.2 INFORMATION FLOW TYPE-CHECKER

This section describes our implementation, called Information Flow Type-checker (IFT), and the type system it enforces. IFT guarantees that if a program is well typed, no information flows exist in the program beyond those expressed in the flow policy that expresses the high-level specification. IFT is sound and conservative: if IFT approves a program, then the program has no undesired information flows, but if IFT issues a warning, then the program might or might not actually have undesired information flows at run time. The guarantee is modulo human examination of a small number of declassifications, including ones about implicit information flow through conditionals.

As shown in Fig. 2.1, a programmer using IFT provides two kinds of information about the information flows in the program. First, the programmer provides a flow policy file, which describes the types of information flows that are permitted in the program (see Sect. 3.2.3). For example, a simple app for recording audio to the file system would have a flow policy containing only `RECORD_AUDIO→FILESYSTEM`. It would be suspicious if this app's flow policy contained `RECORD_AUDIO→INTERNET`, because that flow allows audio to be leaked to an attacker's server.

Second, the programmer writes Java type annotations to express information-flow type qualifiers. Each qualified type includes a set of sensitive sources from which the data may have originated and a set of sinks to which the data may be sent. For example, the programmer of the audio recording app would annotate the type of the recorded data with `@Source(RECORD_AUDIO) @Sink(FILESYSTEM)`. IFT uses type-checking over an information flow type system to verify that the annotated code is consistent with the flow policy.

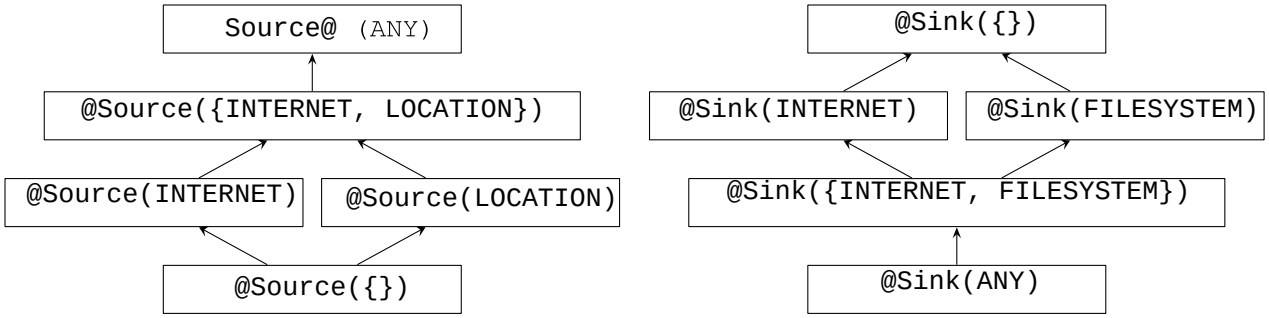

Figure 2 Partial qualifier hierarchy for source and sink type qualifiers `@Source` and `@Sink`.

3.2.1 Types: sources and sinks

The type qualifier `@Source` on a variable's type indicates what sensitive sources might affect the variable's value. The type qualifier `@Sink` indicates where (information computed from) the value might be output. These qualifiers can be used on any occurrence of a type, including in type parameters, object instantiation, and cast types.

As an example, consider the following declaration:

```
@Source(LOCATION) @Sink(INTERNET) double loc;
```

The type of variable loc is `@Source(LOCATION)@Sink(INTERNET)double`. The type qualifier `@Source(LOCATION)` indicates that the value of `loc` might have been derived from location information. The type qualifier `@Sink(INTERNET)` indicates that `loc` might be output to the network.

The arguments to `@Source` and `@Sink` are permissions drawn from our enriched permission system (Sect. 3.2.2). The argument may be a set of permissions to indicate that a value might combine information from multiple sources or flow to multiple locations. The special constant ANY denotes the set of all sources or the set of all sinks; the empty set denotes the absence of sources or sinks.

3.2.1.1 Subtyping

Adding type qualifiers to the Java type system only requires extending the subsumption rule in a standard way; other Java typing rules remain unchanged. A type qualifier hierarchy indicates which assignments, method calls, and overridings are legal, according to standard object-oriented typing rules. **Figure 2** shows parts of the `@Source` and `@Sink` qualifier hierarchies.

`@Source(B)` is a subtype of `@Source`(*A*) iff *B* is a subset of *A* [16]. For example, `@Source(INTERNET)` is a subtype of `@Source({INTERNET, LOCATION})`. This rule reflects the fact that the `@Source` qualifier places an upper bound on the set of sensitive sources that were actually used to compute the value. If the type of x is qualified by `@Source({INTERNET, LOCATION})`, then the value in x might have been derived from both `INTERNET` and `LOCATION` data, or only from `INTERNET`, or only from `LOCATION`, or from no sensitive source at all.

The opposite rule applies for sinks: `@Sink`(*B*) is a subtype of `@Sink`(*A*) iff *A* is a subset of *B*. For example, the type `@Sink({INTERNET, FILESYSTEM})` indicates that the value is permitted to flow to both `INTERNET` and `FILESYSTEM`. This is a subtype of

`@Sink(INTERNET)`, as the latter type provides fewer routes through which the information may be leaked.

Based on these rules, the top type qualifiers of these hierarchies are `@Source(ANY)` and `@Sink({})`, and the bottom type qualifiers are `@Source({})` and `@Sink(ANY)`.

3.2.1.2 Polymorphism

Information flow type qualifiers interact seamlessly with parametric polymorphism (Java generics). For example, a programmer can declare

```
List<@Source(CONTACTS) @Sink(WRITE_SMS) String> myList;
```

to indicate that the elements of `myList` are strings that are obtained from `CONTACTS` and that may flow to `WRITE_SMS`.

IFT also supports qualifier polymorphism, in which the type qualifiers can change independently of the underlying Java type. This allows a programmer to write a generic method that can operate on values of any information flow type and return a result of a different Java type with the same sources/sinks as the input. It also enables qualifier polymorphism even for non-generic Java methods. For example, the method `@PolySource int f(@PolySource int x)` can be passed an `int` with any sources, and the result has exactly the same sources as the input. This qualifier polymorphism can be viewed as the declaration and two uses of a type qualifier variable. The implicit type qualifier variable is automatically instantiated by IFT at the point of use. Given variable `netarg` of type `@Source(INTERNET) int`, in an invocation `f(netarg)` the type qualifier variable is instantiated to `@Source(INTERNET)` and the return type of this method invocation is therefore `@Source(INTERNET) int`.

Polymorphism allows IFT to be context-sensitive.

3.2.2 Comparison to Android permissions

IFT's permission model differs from the Android permission model in three ways. (1) IFT's permissions are statically guaranteed at compile time, whereas Android permissions are enforced at run time, potentially resulting in an exception during execution. If an app inherits a permission from another app with the same `sharedUserId`, IFT requires that permission to be listed in the flow policy. (2) IFT's permission flows are finer-grained than standard Android manifest permissions. Android permits any flow between any pair of permissions in the manifest — that is, an Android program may use any resource mentioned in the manifest in an arbitrary way. (3) IFT refines Android's permissions, as discussed in this section.

3.2.2.1 Sinks and sources for additional resources

IFT adds additional sources and sinks to the Android permissions. For example, IFT requires a permission to retrieve data from the accelerometer, which can indicate the user's physical activity, and to write to the logs, which a colluding app could potentially read. Table 2.1 lists the additional sources and sinks. We selected and refined these by examining the Android API and Android programs, and it is easy to add additional ones. Our system does not add much complexity — it only adds 26 (18%) to the 145 Android permissions.

Some researchers feel that the Android permission model is already too complicated for users to understand [27], but our perspective is that of a full-time auditor who is trained to analyze applications. The flow policy is examined once per application by that skilled engineer, not on every download by a user, so the total human burden is less (Sect. 4.1.3.1 provides empirical measurements). The more detailed flow policy yields more insight than standard Android permissions, because the flow policy makes clear how each resource is used, not just that it is used.

We now discuss two permissions, LITERAL and CONDITIONAL, whose meaning may not be obvious.

Table 1 Additional sources and sinks used by IFT, beyond the built-in 145 Android permissions.

Sources	Sinks	Both source and sink
ACCELEROMETER	CONDITIONAL	CAMERA_SETTINGS
BUNDLE	DISPLAY	CONTENT_PROVIDER
LITERAL	SPEAKER	DATABASE
MEDIA	WRITE_CLIPBOARD	FILESYSTEM
PHONE_NUMBER	WRITE_EMAIL	PARCEL
RANDOM	WRITE_LOGS	PROCESS_BUILDER
READ_CLIPBOARD		SECURE_HASH
READ_EMAIL		SHARED_PREFERENCES
READ_TIME		SQLITE_DATABASE
USER_INPUT		SYSTEM_PROPERTIES

Literal The LITERAL source is used for programmer-written constants (in the source code, Android manifest, or resource files) such as `"Hello world!"`, and for any variable whose value is computed using only those constants. This enables IFT to distinguish information derived from the program source code from other inputs. Program literals are not trusted, since the app vendor may be malicious. The flow policy shows how they are used in the program.

Conditional The CONDITIONAL sink is used for conditional expressions — every value used in a conditional expression flows to that sink. This enables IFT to raise a warning at locations where the control flow of the program branches on sensitive information. The auditor reviews those warnings to detect implicit information flows, as explained in greater detail in Sect. 3.2.7.

3.2.2.2 Restricting existing permissions

The standard Android permissions might be too coarse-grained to express the developer's intention. For example, Android's INTERNET permission represents all reachable hosts on the Internet. IFT allows this permission to be parameterized with a domain name, as in INTERNET("*.google.com"). Other permissions can be parameterized in a similar style, and the meaning of the optional parameter varies based on the permission it refines. For example, a parameter to FILESYSTEM represents a file or directory name or

wildcard, whereas the parameter to SEND_SMS represents the phone number that receives the SMS. Other permissions that can be parameterized include CONTACTS, *_EXTERNAL_FILESYSTEM, NFC,

*_SMS, and USE_SIP. Several of the additional sources and sinks (Table 1) can also be parameterized, such as USER_INPUT to distinguish sensitive from non-sensitive user input.

IFT performs intraprocedural constant value propagation to enable precise analysis of parameterized permissions.

3.2.3 Flow policy

A flow policy is a list of all the information flows that are permitted to occur in an application. A flow policy file expresses a flow policy, as a list of flowsource → flowsink pairs. Just as the Android manifest lists all the permissions that an app uses, the flow policy file lists the flows among permissions and other sensitive locations.

Consider the "Block SMS" application of Table 3, which blocks SMS messages from a blacklist of blocked numbers and saves them to a file for the user to review later. Its flow policy must contain READ_SMS→FILESYSTEM to indicate that information obtained using the READ_SMS permission is permitted to flow to the file system.

The flow policy specifies what types are legal Every flow in a program is explicit in the types of the program's expressions. For example, if there is no expression whose type has the type qualifiers `@Source(CAMERA) @Sink(INTERNET),` then the program never sends data from the camera to the Internet (modulo conditionals and transitive flows). The expression's type might be written by a programmer or might be automatically inferred by IFT.

IFT guarantees that there is no information flow except what is explicitly permitted by the flow policy. If the type of a variable or expression indicates a flow that is not permitted by the flow policy, then IFT issues a warning even if the program otherwise would type-check. For example, the following declaration type-checks, but IFT would still produce an error unless the flow policy permits the CAMERA→INTERNET flow:

```
@Source(CAMERA) @Sink(INTERNET) Video video = getVideo();
```

Transitive flows Transitive flows through on-device source-sink pairs must be explicitly written in the flow policy. This is because apps can use on-device sinks to whitewash sensitive information. For example, if a flow policy permits USER_INPUT→FILESYSTEM and FILESYSTEM→INTERNET, then an application might write user input to a file and then send the contents of that file to a malicious server. Therefore, the transitive flow USER_INPUT→INTERNET must be explicitly stated in the flow policy.

Parameterized permissions (Sect. 3.2.2.2) reduce the number of transitive flows. For example, if user input is only written to files in the notes directory (USER_INPUT→FILESYSTEM("notes/*")INTERNET) and only), then files in the cat-photos directory are sent to the Internet (FILESYSTEM("cat-photos/*")→ the transitive flow USER_INPUT→FILESYSTEM is not required.

On-device source-sink pairs involving resources that may be accessed by other apps could be used by colluding apps to leak information. To prevent this, the flow policies of all apps on a device or in an app store are checked against each other for inter-app transitive flows. If a transitive flow is found that violates a flow policy of an app, then one or more apps may need to be excluded or rewritten. In practice, app stores will specify standard policies for flows including these source-sink pairs, so that developers can avoid writing conflicting apps.

An off-device sink, such as a website or the recipient of an SMS, might leak data to some sink not allowed by the flow policy. Off-device sinks must be either trusted or verified by other means.

3.2.4 Inference and defaults

A complete type consists of a `@Source` qualifier, a `@Sink` qualifier, and a Java type. To reduce programmer effort and code clutter, most of the qualifiers are inferred or defaulted rather than written as type annotations. A programmer need not write type annotations within method bodies, because such types are inferred by IFT. For method signatures and fields, a programmer generally writes either

`@Source` or `@Sink`, but not both. We now explain the inference and defaulting features.

3.2.4.1 Type inference and flow-sensitivity

A programmer does not write information flow types within method bodies. Rather, local variable types are inferred.

IFT implements this inference via flow-sensitive type refinement. Each local variable declaration (also casts and resource variables) defaults to the top type qualifiers, `@Source(ANY) @Sink({})`. At every properly-typed assignment statement, the type of the left-hand side is flow-sensitively refined to that of the right-hand side, which must be a subtype of the left-hand side's declared type. The refined type applies until the next side effect that might invalidate it.

Consider the following simple method:

```
void process(@Source(INTERNET) int netint,
             @Source(LOCATION) int locint) {
  int x; // x is defaulted to @Source(ANY) @Sink({}) int
  x = netint;  // x is refined to @Source(INTERNET)
  int x = locint;    // x is refined to @Source(LOCATION) int
}
```

Flow-sensitive type refinement spares the programmer from writing type qualifiers on local variable `x`, and the system automatically determines the most precise type in each context.

IFT limits type inference to method bodies to ensure that each method can be type-checked in isolation, with a guarantee that the entire program is type-safe if each method has been type-checked. It would be possible to perform a whole-program type inference, but such an approach would be heavier-weight, would need to be cognizant of cooperating or communicating applications, could cause a change in one part of a program to cause new type-checking errors elsewhere, and would provide fewer documentation benefits.

3.2.4.2 Determining sources from sinks and vice versa

If a type is annotated with only a source or only a sink, the other qualifier is filled in with the most general value that is consistent with the flow policy. If the programmer writes `@Source`(α), IFT defaults this to `@Source`(α) `@Sink`(ω) where ω is the set of sinks that all sources in α can flow to. Similarly,

`@Sink`(ω) is defaulted to `@Source`(α)`@Sink`(ω) where α is the set of sources allowed to flow to all sinks in ω. Defaults are not applied if the programmer writes both a source and a sink qualifier.

Suppose the flow policy contains the following:

```
CAMERA -> DISPLAY,DATABASE
LOCATION -> DATABASE
```

Table 2 Default information flow qualifiers for unannotated types.

Location	Default information flow qualifier
Method parameters & receivers	`@Sink(CONDITIONAL)`
Return types	`@Source(LITERAL)`
Fields	`@Source(LITERAL)`
`null`	`@Source({}) @Sink(ANY)`
Other literals	`@Source(LITERAL)`
Type arguments	`@Source(LITERAL)`
Upper bounds	`@Source(ANY) @Sink({})`
Local & resource variables	`@Source(ANY) @Sink({})`

Then these pairs are equivalent:

```
@Source({LOCATION}) = @Source({LOCATION}) @Sink(DATABASE)
@Sink(DATABASE) = @Source({CAMERA,LOCATION}) @Sink(DATABASE)
```

This mechanism is useful because oftentimes a programmer thinks about a computation in terms of only its sources or only its sinks. The programmer should not have to consider the rest of the program that provides context indicating the other end of the flow.

An example of a method that uses only a `@Source` qualifier is the File constructor: a newly-created readable file should be annotated with `@Source(FILESYSTEM)`, but there is no possible `@Sink` qualifier that would be correct for all programs. Instead, the `@Sink` qualifier is omitted, and our defaulting mechanism provides the correct value based on the application's flow policy.

This defaulting mechanism is essential for annotating libraries. We wrote manual annotations for

10,470 methods of the Android standard library. Only 7 of the API methods annotated so far use both a `@Source` and a `@Sink` qualifier. For example,

```
Camera.setPreviewDisplay(
@Source(CAMERA) @Sink(DISPLAY) SurfaceHolder holder)
```

The parameter `holder` both receives photos from the camera and displays them.

This mechanism can be viewed as another application of type polymorphism: defaulting of types depends on the flow policy and the same source code can be reused in different scenarios by using a different flow policy.

3.2.4.3 Defaults for unannotated types

Table 2 shows the default qualifiers for completely unannotated types. When the default is only a source or only a sink, the other qualifier is inferred from the flow policy as explained in Sect. 3.2.4.2.

Most unannotated types (including field types, return types, generic type arguments, and non-`null` literals) are given the qualifier `@Source(LITERAL)`. This is so that a simple computation involving only constants does not require annotations.

As is standard, the `null` literal is given the bottom type qualifiers `@Source({})` `@Sink(ANY)`, enabling an assignment to any variable.

3.2.5 Declassifications

Every sound static analysis is conservative: that is, there exists source code that never misbehaves at run time, but the static analysis cannot prove that fact and issues a warning about possible misbehavior. Every downcast in a Java program is an example of such conservatism in the Java type system. In the context of information flow analyses, an example would be a database: in general, a database query can return arbitrary sensitive data, but application invariants might guarantee that a particular query always returns non-sensitive data. IFT would warn about use of any database query result in a context that could leak the result, but in the example the warning would be a false positive.

In order to suppress a warning that is a false positive, the developer declassifies data that was typed too conservatively using a downcast. The developer is required by the app store to write a justification for each declassification. The app store auditor manually verifies both the justification and the declassification. Thus, the auditor validates the developer's claim that the code is well-behaved for some reason that is beyond the precision of the type checker.

In 11 Android apps (9437 LOC), IFT suffered 26 false positives, or fewer than 3 per 1,000 LOC.

3.2.6 Indirect control flow

Indirect control flow, for example in reflection, intents, or exception handling, is challenging for a static analysis. IFT soundly handles these constructs through additional analyses and conservative assumptions.

IFT analyzes Java reflection to determine the target method of a reflective call. This enables a downstream analysis, such as IFT's information-flow type-checking, to treat the reflective code as a direct method call, which has a much more precise annotated signature than does `Method.invoke`. IFT's analysis resolves the reflective call to a single concrete method in 96% of cases in our experiments, including malicious examples where reflection is used intentionally as a form of code obfuscation. The library's conservative annotations for `Method.invoke` ensure that any unresolved reflective call is treated soundly.

Intents are an Android mechanism for interprocess communication, and they can also create processes (Android activities). To handle intents, we extended IFT with map types (similar to record types) that represent the mappings of data in an intent payload. Each app implements intent-receiving methods, and their type signatures act as interface specifications and permit modular checking. As long as new apps are consistent with annotations on previously-checked apps that they may communicate with, the old apps need not be re-checked.

IFT soundly handles other indirect control flows, such as exception handling. For example, types in catch clauses are enforced to be supertypes of any exception they may catch.

3.2.7 Implicit information flow

Implicit information flow through conditionals can leak private information. For example, consider the following code and a flow policy containing LITERAL→INTERNET:

```
@Source(USER_INPUT) long creditCard = getCC();
final long MAX_CC_NUM 9999999999999999;
for (long i = 0 ; i < MAX_CC_NUM ; i++) {
  if (i == creditCard) sendToInternet(i);
}
```

This code leaks the credit card number to the Internet using the flow LITERAL→INTERNET and the fact that `i` is only sent to the Internet when `i == creditCard` evaluates to true.

The classic approach of Denning and Denning [17] to detect implicit information flow is to taint all computations in the dynamic scope of a conditional statement with all the sources from the conditional's predicate. This includes all statements in the body of the conditional and all statements in any method directly or indirectly called by the body. Over-tainting of computations within the dynamic scope of conditionals leads to many false positive alarms. These alarms occur far from the conditional statement or other statement(s) that caused them. In order to determine whether an implicit information flow truly occurs, the auditor has to work backward from the location of an alarm to the conditional statement or statements that caused it.

In our approach, the auditor reviews every conditional statement that uses a sensitive source in its predicate. The auditor first decides whether the knowledge about the boolean result of the predicate is sensitive information. For example, checking whether a credit card number has 16 digits does not reveal anything sensitive — in this case, the auditor need not review the body of the conditional. However, if the auditor decides that the conditional predicate is sensitive, he/she must rule out any implicit information flow that violates the flow policy. In order to determine whether an implicit information flow truly occurs, the auditor works from the body of the conditional forward to all statements in dynamic scope that might implicitly leak information.

In both the classic approach and our approach, the auditor has to carefully review the dynamic scope of the conditional body to rule out false positives. However, unlike the classic approach, in our approach, the reviewer is aware of the context of the conditional and can make a more informed decision about whether an implicit information flow might occur at runtime.

The auditors in our experiments (Sect.4.1.3.1) felt that our approach was easier for them than the classic one. They preferred to think about an entire conditional expression at once rather than statement-by-statement. Oftentimes, examining a conditional expression enabled the auditors to rule out bad behavior without needing to examine any statement in its dynamic scope; this was particularly true for simple conditionals such as tests against `null`.

3.2.8 Implementation

IFT is implemented as a pluggable type system built on top of the Checker Framework [18] and uses standard Java type annotations. The implementation of IFT consists of 3,731 lines of Java, plus annotations for 10,470 library methods. IFT's source code is available at `http://types.cs. washington.edu/sparta/`. Version 0.9.6 was used for the experiments presented here.

3.2.9 Limitations

IFT is focused on Trojans that cause an undesired information flow, as indicated by the threat model of Sect. 3.1.1.2. IFT should be used in conjunction with complementary techniques that address other security properties. This section discusses further limitations.

As with any static analysis, IFT's soundness guarantee only extends to code that is analyzed at compile time. Use of native code and un-analyzed Android activities requires a different analysis or trusted annotations that describe the information flows induced by those components. IFT currently forbids dynamic code loading, because IFT type-checks source code. Dynamic class loading could be soundly allowed if the loaded classes type-check and their public signatures are the same as were assumed at compile time. To achieve this would require load-time type-checking of compiled (.class or .dex) files. Re-implementing the IFT type rules for binaries would be an engineering challenge, but not a conceptual one.

Our cooperative verification model means that the vendor knows one of the techniques that the app store will use to verify an app. This knowledge might permit a malicious developer to design Trojans that are beyond the capabilities of IFT or that exploit IFT's limitations.

As with many security mechanisms, human judgment can be a weak link. A malicious developer could write a misleading explanation for an information flow in the flow policy or for a declassification, in an effort to convince the auditor to approve malware. Our work does not address how to establish an app store's policies.

Despite these limitations, use of IFT increases the difficulty of hiding Trojans in source code. The requirement that code be accepted by IFT may also make the Trojan more likely to be detected using other tools or manual verification.

3.2.10 Future work

We plan to enrich flow policies in three ways, while retaining the simple and high-level flavor of these specifications. (1) We will refine permissions, such as splitting the WRITE_CONTACTS permission so that separate policies can be specified for email addresses, phone numbers, and notes fields. (2) The flow policy will indicate not just the endpoints of the information flow, but an entire path. For example, it might be valid to send personal information to the Internet only if it has passed through an encryption

module first. (3) The flow policy will indicate conditional information flows, such as permitting information flow from the microphone to the network only when the user presses the “transmit” button.

3.3 STATIC ANALYSIS OF IMPLICIT CONTROL FLOW

Our work improves the precision of a downstream static analysis, by eliminating false positive warnings in cases of implicit control flows. Imprecision due to implicit control flow affects every static analysis.

```
1  class ArticleViewActivity extends Activity {
2    void onCreate(Bundle savedInstanceState) {
3      if (android.os.Build.VERSION.SDK_INT >= 11) {
4      // Android version 11 and later has Action Bar
5      Method getActionBar =
6                              getClass().getMethod("getActionBar");
7      @Low Object actionBar = getActionBar.invoke(this);
8        ...
9      }
10   }
11 }
12
13 // Library annotations:
14 class Method {
15   @High Object invoke(Object obj, Object... args) {...}
16 }
17 class Activity {
18   // Only exists in Android SDK 11 and above.
19   @Low ActionBar getActionBar() {...}
20 }
```

Figure 3: A noninterference type-checker produces a false positive warning on line 7, where the return type of `Method.invoke`, of type `High`, is assigned to variable `actionBar` which has declared type `Low`. The call on line 7 always returns a Low value at run time (even though other calls to `invoke` may in general return a `High` value), so the assignment is safe. When the noninterference type system is augmented by our reflection analysis, it no longer issues the false positive warning.

For concreteness, consider a noninterference type system [84], which guarantees that the program does not leak sensitive data.

The noninterference type system distinguishes high-security-level values from low-security-level values; for brevity, `High` and `Low` values. The static property checked is that values in `High` variables are not assigned to `Low` variables, which could leak sensitive data. Variables and expressions marked `High` may hold a `Low` value at run time; this is also expressed as `Low` <: `High`, where the symbol “<:” denotes subtyping. To use this type system, a user annotates each type with `High` or `Low`, the default being `Low`. The type system is conservative: if it issues no warnings, then the program has no interference and running it does not leak any `High` data to Low contexts.

When run on the Android app `Aard Dictionary` (`http://aarddict.org/`), the noninterference type system issues false positive warnings due to its conservative handling of implicit control flows. When our reflection and intent analyses are integrated into it, the type system remains sound but no longer issues the false positive warnings. The examples in this section use a noninterference type system, but other type systems suffer similar false positives. Our reflection and intent analyses also help other downstream analyses, as demonstrated in Section 4.4.4.

3.3.1 Reflection

Some calls to `Method.invoke` return a `High` value at run time. Thus, the signature of `Method.invoke` (line 15 of Figure 3) must have a `High` return type; any other return type in the summary would be unsound. Some calls to `Method.invoke` always return a `Low` value. The conservative signature of `Method.invoke` causes false positive warnings in such cases.

Figure 3 illustrates the problem in `Aard Dictionary`. The component `ArticleViewActivity` uses an

```
1     class DictionaryMain extends Activity {
2     void translateWord(int source, int target, String word){
3     Intent i = new Intent(this, WordTranslator.class);
4     i.putExtra("source", source);
5     i.putExtra("target", target);
6     i.putExtra("word", word);
7     startActivity(i);
8     }
9     }
10
11    class WordTranslator extends Activity {
12    void onCreate(Bundle savedInstanceState)
13    Intent i = getIntent();
14    @Low int source = i.getIntegerExtra("source");
15    @Low int target = i.getIntegerExtra("target");
16    @Low String word = i.getStringExtra("word");
17    showResult(translate(source, target, word));
18    }
19    String translate(int source, int target, String word) {...}
20    Intent getIntent() {...}
21    void showResult(String result) {...}
22    }
23
24    // Library annotations:
25    class Intent {
26    @High Integer getIntegerExtra(String key) {...}
27    @High String getStringExtra(String key) {...}
28    }
```

Figure 4: A noninterference type-checker produces false positive warnings on lines 14–16, where the return type of `get*Extra`, of type `High`, is assigned to variables with declared type `Low`. The calls on lines 14–16 always return a `Low` value at run time (even though other calls to `get*Extra` may in general return a `High` value), so the assignments

are safe. When the noninterference type system is augmented by our intent analysis, it no longer issues the false positive warnings.

`ActionBar`, which is a feature that was introduced in version 11 of the Android API. In order to prevent run-time errors for a user who has an older version of Android (and also to enable the app to compile when a developer is using an older version of the Android API), this app uses reflection to call methods related to the `ActionBar`. The noninterference type-checker issues a false positive due to the use of reflection; our reflection analysis (Section 3.4) eliminates the false positive warning.

3.3.2 Android intents

An Android component might send a `High` value via an intent message to another component; therefore, the summary for methods that retrieve data from an intent (lines 26–27 of Figure 4) must conservatively assume that the data is a `High` value. This conservative summary may cause false positive warnings when the data is of type `Low` at run time.

Figure 4 shows another example from `Aard Dictionary`. The components `DictionaryMain` and

`WordTranslator` use Android intents to communicate. Android intents are messages sent between Android components, and those messages contain "extras", which is a mapping of keys to objects. Component `DictionaryMain` creates an intent object `i`, adds Low-security extra data to `i`'s extras mapping, and on line 7 calls the Android library method `startActivity` to send the intent. The Android system then calls `WordTranslator.onCreate`, which is declared on line 12. The noninterference type-checker issues a false positive due to the use of intents; our intent analysis (Section 3.5) eliminates the false positive warning.

3.4 REFLECTION RESOLUTION

Reflection is a metaprogramming mechanism that enhances the flexibility and expressiveness of a programming language. Its primary purpose is to enable a program to dynamically exhibit behavior that is not expressed by static dependencies in the source code.

Reflection is commonly used for the following four use cases, among others. (1) Provide backward compatibility by accessing an API method that may or may not exist at run time. The reflective code implements a fallback solution so the app can run even if a certain API method does not exist, e.g., on older devices. (2) Access private API methods and fields, which offer functionality beyond what is provided by the public API. (3) Implement design patterns such as duck typing. (4) Code obfuscation to make it harder to reverse-engineer the program, e.g., code that accesses premium features that require a separate purchase. The Android developer documentation encourages the use of reflection to provide backward compatibility and for code obfuscation (cases 1 and 4 above), and 39% of apps in the F-Droid repository [26] use reflection.

Not all uses of reflection can be statically resolved, but our experiments show that many of them can. Whenever the developer runs a code analysis, it is beneficial to the analysis if as much reflection as possible is resolved, in order to reduce false positive warnings. Obfuscation is not compromised, because analysis results, annotations, and

other information that is used in-house by the developer need not be provided to users of the software.

Approach for reflection resolution

Without further information about what method is reflectively called, a static analysis must assume that a reflective call could invoke any arbitrary method. Such a conservative assumption increases the likelihood of false positive warnings.

At each call to `Method.invoke`, our analysis soundly estimates which methods might be invoked at runtime. Based on this estimate, our analysis statically resolves the `Method.invoke` call — that is, it provides type information about arguments and return types for a downstream analysis. The results are soundly determined solely based on information available at compile time.

The reflection resolution consists of the following parts: *Reflection type system*: Tracks and infers the possible names of classes, methods, and constructors used by reflective calls. (Section 3.4.1)

Reflection resolver: Uses the reflection type system to estimate the signatures of methods or constructors that can be invoked by a reflective call. (Section 3.4.2)

3.4.1 Reflection type system

Our reflection type system refines the Java type system to provide more information about array, `Class`, `Method`, and `Constructor` values. In particular, it provides an estimate, for each expression of those types, of the values they might evaluate to at run time.

For arrays, the refined type indicates the length of the array: for example, `@ArrayLen({3,4})` indicates that the array will be of length 3 or 4. For expressions of type `Class`, there are two possible type qualifiers, `@ClassVal` and `@ClassBound`, representing either an exact `Class` value or an upper bound of the `Class` value. The list of possible values is expressed as an array of strings representing fully-qualified types; for example, `@ClassVal("java.util.HashMap")` indicates that the `Class` object represents the `java .util.HashMap` class. Alternatively, `@ClassBound("java.util.HashMap")` indicates that the Class object represents java.util.HashMap or a subclass of it.

For expressions of type `Method` and `Constructor`, the type qualifier indicates estimates for the class, method name, and number of parameters. For example,

@MethodVal(cn="java.util.HashMap", mn={"containsKey", "containsValue"}, np=1)

indicates that the method represents either `HashMap.containsKey` or `HashMap.containsValue`, with exactly 1 parameter. Likewise, the `MethodVal` type may have more than one value for the class name or number of parameters. The represented methods are the Cartesian product of all possible class names, method names, and numbers of parameters. For a constructor, the method name is "`<init>`", so no separate `@ConstructorVal` type qualifier is necessary.

The `MethodVal` type is imprecise in that it indicates the number of parameters that the method takes, but not their type. This means that the type system cannot distinguish methods in the uncommon and discouraged [10] case of method overloading. This was a conscious design decision that reduces the verbosity and complexity of the annotations, without any practical negative consequences. In our experiments with more than 300,000

lines of Java code, this imprecision in the type system never prevented a reflective call from being resolved.

Our implementation caps the size of a set of values at 10. This cap was never reached in our case studies. If a programmer writes, or the type system infers, a set of values of size larger than 10, then the type is widened to its respective top type. A top type indicates that the type system has no estimate for the expression: the type system's estimate is that the run-time value could be any value that conforms to the Java type. The top type is the default, and it is represented in source code as the absence of any annotation.

3.4.1.1 Type checking

The reflection type system enforces standard type system guarantees, e.g. that the right-hand side of an assignment is a subtype of the left-hand side. These typing rules follow those of Java, they are standard for an object-oriented programming language, and they are familiar to programmers. Therefore, we do not detail them in this document. The reflection type system and our implementation are compatible with all Java features, including generics (type polymorphism).

3.4.1.2 Type inference

Programmers do not need to write type annotations within method bodies, because our system performs local type inference. More specifically, for local variables, casts, and instanceof expressions, the absence of any annotation indicates that the type system should infer the most precise possible type from the context. For all other locations — notably fields, method signatures, and generic type arguments — a missing annotation is interpreted as the top type qualifier.

$$\frac{e : \texttt{String} \quad val \text{ is the statically computable value of } e}{e : \texttt{@StringVal(}val\texttt{)}}$$

$$\frac{e : \texttt{int} \quad val \text{ is the statically computable value of } e}{e : \texttt{@IntVal(}val\texttt{)}}$$

$$\frac{e : \texttt{@IntVal(}\pi\texttt{)}}{\texttt{new C[}e\texttt{]} : \texttt{@ArrayLen(}\pi\texttt{)}}$$

$$\frac{}{\texttt{new C[]\{}e_1\texttt{,...,}e_n\texttt{\}} : \texttt{@ArrayLen(}n\texttt{)}}$$

Figure 5: Inference rules for `@StringVal`, `@IntVal`, and `@ArrayLen`.

$$\frac{fqn \text{ is the fully-qualified class name of } \texttt{C}}{\texttt{C.class} : \texttt{@ClassVal(}fqn\texttt{)}}$$

$$\frac{s : \texttt{@StringVal(}\nu\texttt{)}}{\texttt{Class.forName(}s\texttt{)} : \texttt{@ClassVal(}\nu\texttt{)}}$$

$$\frac{fqn \text{ is the fully-qualified class name of the static type of } e}{e\texttt{.getClass()} : \texttt{@ClassBound(}fqn\texttt{)}}$$

$$\frac{\begin{array}{c}(e : \texttt{@ClassBound(}\nu\texttt{)} \ \vee \ e : \texttt{@ClassVal(}\nu\texttt{)}) \\ s \ : \ \texttt{@StringVal(}\mu\texttt{)} \quad p \ : \ \texttt{@ArrayLen(}\pi\texttt{)}\end{array}}{e\texttt{.getMethod(}s\texttt{,}p\texttt{)} : \texttt{@MethodVal(cn=}\nu\texttt{,mn=}\mu\texttt{,np=}\pi\texttt{)}}$$

$$\frac{e : \texttt{@ClassVal(}\nu\texttt{)} \quad p : \texttt{@ArrayLen(}\pi\texttt{)}}{e\texttt{.getConstructor(}p\texttt{)} : \texttt{@MethodVal(cn=}\nu\texttt{,mn="<init>",np=}\pi\texttt{)}}$$

Figure 6: Selected inference rules for the `@ClassVal`, `@ClassBound`, and `@MethodVal` annotations. Additional rules exist for expressions with similar semantics but that call methods with different names or signatures, and for fields/returns.

The local type inference is flow-sensitive. It takes advantage of expression typing rules that yield more precise types than standard Java type-checking would.

Estimates for values of expressions We have designed and implemented a dataflow analysis that infers and tracks types providing an estimate for the possible values of each expression. Our implementation goes beyond constant folding and propagation: it evaluates side-effect-free methods, it infers and tracks the length of each array, and it computes a set of values rather than just one. For example, `@ArrayLen({3,4})` indicates that at run time the array has length 3 or 4. **Figure 5** shows selected inference rules. The reflection type system builds on top of this dataflow analysis.

Inference of `@ClassVal` and `@ClassBound` The reflection type system infers the exact class name (`@ClassVal`) for a Class literal (`C.class`), and for a static method call (e.g., `Class.forName(arg)`, `ClassLoader.loadClass(arg)`, ...) if the argument has a sufficiently precise `@StringVal` estimate. In contrast, it infers an upper bound (`@ClassBound`) for instance method calls (e.g., `obj.getClass()`).

An exact class name is necessary to precisely resolve reflectively-invoked constructors since a constructor in a subclass does not override a constructor in its superclass. Either an exact class name or a bound is adequate to resolve reflectively-invoked methods because of the subtyping rules for overridden methods.

Inference of `@MethodVal` The reflection type system infers `MethodVal` types for methods and constructors that have been created via Java's Reflection API. A nonexhaustive list of examples includes calls to `Class.getMethod(String name, Class<?>... paramTypes)` and `Class.getConstructor(Class<?>... paramTypes)`. For example, the type inferred for variable `getActionBar` on line 5 of Figure 3.1 is

```
@MethodVal(cn="ArticleViewActivity", mn="getActionBar", np=0).
```

Although Figure 3.1 uses raw (non-parameterized) types, our inference supplies the missing type argument information.

Inference of field types For private fields, our type inference collects the types of all assignments to the field, and sets the field type to their least upper bound (lub). If the lub is not a subtype of the declared type, this step is skipped and a type-checking error will be issued at some assignment. The same mechanism works for non-private fields, but the entire program has to be scanned for assignments. At the end of type-checking, the type-checker outputs a suggestion about the field types. The user may accept these suggestions and re-run type-checking to obtain more precise results; we did so in our experiments. Field type inference works for every type system, not just those related to reflection.

Method signature inference Similarly to field type inference, private method parameters are set to the lub of the types of the corresponding arguments, and private method return types are set to the lub of the types of all returned expressions, when those are consistent with the declared types. For non-private methods, the entire program is scanned for calls/overriding and the type-checker outputs suggestions.

Figure 6 shows selected inference rules for the reflection type system.

3.4.2 Reflection resolver

Prior work (see Section 4.6.4) commonly re-writes the source code or changes the AST within the program analysis tool, changing a call to `Method.invoke` into a call to the method that is reflectively invoked before analyzing the program. This approach interferes with the toolchain, preventing the code from being compiled or run in certain environments. This approach is also at odds with the very purpose of reflection: the program no longer adapts to its run-time environment and loses properties of obfuscation. A final problem is that an analysis may discover facts that cannot be expressed in source code form.

Our reflection resolver operates differently: it leaves the program unmodified but narrows the procedure summary — the specification of parameter and return types used during modular analysis — for that particular call site only. When the downstream analysis requests the summary at a call to `Method.invoke`, it receives the more precise information rather than the conservative summary that is written in the library source code. This transparent integration means that the downstream analysis does not need to be changed at all to be integrated with the reflection analysis.

3.4.2.1 Example

Recall the example of Figure 3. When the noninterference type system analyzes `getActionBar.invoke(this)` on line 7, it uses a method summary (like a declaration) to indicate the requirements and effects of the call. Ordinarily, it would use the following conservative declaration for `Method.invoke`:

```
@High Object invoke(Object recv, Object ... args)
```

However, the reflection type system inferred that the type of variable `getActionBar` is `@MethodVal(cn="ArticleViewActiv mn="getActionBar", np=0)`. In other words, at run time, the invoked method will be the following one from class `ArticleViewActivity`:

```
@Low ActionBar getActionBar ()
```

Thus, the noninterference type system has a precise type, `Low`, for the result of the invoke call. The reflection resolver provides the following precise procedure summary to the downstream analysis, for this call site only:

```
@Low Object invoke(Object recv, Object ... args)
```

As a result, the type system does not issue a false positive warning about the assignment to variable `actionBar` on line 7.

The summary contains not just refined procedure return types as shown above, but also refined parameter types, enabling a downstream analysis to warn about clients that pass arguments that are not legal for the reflectively-invoked method. It would be possible to refine the Java types as well as the type qualifiers (for instance, to warn about possible run-time type cast errors or to optimize method dispatch), but our implementation does not do so.

If the reflectively-called method or constructor cannot be resolved uniquely, the reflection resolver determines the least upper bound of all return values and the greatest lower bound of all parameter and receiver types.

3.5 ANDROID INTENT ANALYSIS

An Android app is organized as a collection of components that roughly correspond to different screens of an application and to background services.[2] Some apps consist of dozens of components. Intents are used for inter-component communication, both within an app and among different apps. Intents are similar to messages, communicated asynchronously across components. Sending an Android intent implicitly invokes a method on the receiving component, just as making a reflective procedure call implicitly invokes a method. The use of intents is prevalent in Android apps: all top 50 popular paid apps and top 50 popular free apps from the Google Play store use intents [14], the top 838 most popular apps contain a total of 58,989 inter-component communication locations [64], and intents are a potential target for attackers to introduce malware [14].

Intents present two challenges to static analyses: (i) control flow analysis, or determining which components communicate with one another, and (ii) data flow analysis, or determining what data is communicated. Both parts are important. An existing analysis, Epicc [64], partially solves the control flow challenge. Section 3.5.1 describes how our implementation uses Epicc to compute component communication. Our key research contribution is to address the data flow challenge, which has resisted previous researchers. Section 3.5.2 presents a novel static analysis that estimates the data passed in an Android intent.

The structure of Android intents

In addition to attributes that specify which components may receive the intent, an intent contains a map from strings to arbitrary data, called “extras”. The extras map is used to pass additional information that is needed to perform an action. For example, an intent used to play a song contains the song’s title and artist as extras. An invocation of the `putExtra` method adds a key–value entry to the intent map, which can be looked up

[2] Activity, Service, BroadcastReceiver, and ContentProvider are the four kinds of Android components. See http:
//developer.android.com/guide/components/fundamentals.html#Components.

via the `getExtra` method call. Without loss of generality, we will consider that every intent attribute is an entry in the map of extras. The use of extras is prevalent in Android: of the 1,052 apps in the F-Droid repository [26], 69% use intents with extra data. Figure 3.2 shows the common use case of an Android app sending and receiving an intent containing extras.

3.5.1 Component communication patterns

To precisely analyze the types of data sent through intents, our analysis requires `sendIntent` calls to be matched to the declarations of `onReceive` methods they implicitly invoke. We express this matching as a component communication pattern (CCP): a set of pairs of the form (`sendIntent`(*a*,*i*), `onReceive`(*b*,*j*)). Each pair in the CCP indicates that components *a* and *b*, possibly from different apps, may communicate through intents *i* and *j*, which intuitively denote the actual arguments and formal parameters of the implicit invocation.

To precompute an approximated CCP, our current implementation uses APKParser [4], Dare [62], and Epicc [64]. Our implementation inherits Epicc's limitations. Note, however, that Epicc's limitations are not inherent to our intent analysis, and they would disappear if we used a better analysis to compute CCP. As better CCP techniques become available, they can be plugged into our implementation. IC3 [63] is Epicc's successor, created by the same research group. We attempted to use IC3, but we discovered a soundness bug: dynamically-registered Broadcast Receivers were not being analyzed. The IC3 authors have confirmed but not fixed the bug[3], so we used Epicc instead. We now discuss sources of imprecision and unsoundness due to Epicc.

Epicc's sources of imprecision Epicc's lack of support for URIs leads to imprecision since intents with the same action and category but different URIs are conservatively considered equal. As expected of a static analysis, Epicc also cannot handle cases where dynamic inputs determine the identity of receiver components. Epicc also handles this conservatively: all components are considered possible receivers. Furthermore, the points-to and string analyses used by Epicc are also sources of imprecision.

Even with these limitations, all mentioned in [64], Epicc reports 91% precision in a case study with 348 apps.

Epicc's sources of unsoundness Epicc unsoundly assumes that Android apps use no reflection. We used the type system of Section 3.4.1 to circumvent this limitation; see Section 4.4. Epicc also unsoundly assumes that Android apps use no native calls, a standard limitation of static analysis that is shared by IC3. We do not circumvent this limitation. Another unsoundness is the closed-world assumption; that is, Epicc assumes that it knows all the apps installed on a device. Our work shares this assumption. Compatibility with Epicc's ananlysis could be checked whenever an app is installed.

Recall that while finding CCP is necessary, it is not sufficient. Since acceptable solutions exist for finding CCP, the focus of our intent analysis is the unsolved problem of estimating the payloads of intents, which is discussed below.

[3] `https://github.com/siis/ic3/issues/1`

3.5.2 Intent type system

This section presents a type system for Android intents. The type system verifies that the type of data stored within an intent conforms to the declared type of the intent, even in the presence of implicit invocation via intents.

For simplicity, this document abstracts all methods that send intents as the method `sendIntent`, and all methods that receive an intent as the method `onReceive`. For example, in Figure 3.2, `startActivity()`, called on line 7, is an example of a `sendIntent` method, and the method `getIntent()`, declared on line 20, is an example of an `onReceive` method.

The type system verifies that for any `sendIntent` method call and any `onReceive` method declaration that can be invoked by the call site, the intent type of the argument in the `sendIntent` call is compatible with the intent type of the parameter declared in the `onReceive` method signature.

3.5.2.1 Intent types

We introduce intent types, which hold key–type pairs that limit the values that can be mapped by a key.

Syntax of intent types This document uses the following syntax for an intent map type:

```
@Intent("K1" → t1, ..., "Kn" → tn) Intent i = ...;
```

where {*"K1"*,...,*"Kn"*} is a set of literal strings and {*t1*,...,*tn*} is a set of types. The type of variable *i* above consists of a type qualifier *@Intent*(...) and a Java type *Intent*. The regular Java type system verifies the Java type, and our intent type system verifies the type qualifier.

The actual Java syntax used by our implementation is slightly more verbose than that in this document:

```
@Intent(@Entry(key="K1", type="t1"), ...,
   @Entry(key="Kn", type="tn")) Intent i = ...;
```

Semantics of intent types If variable `i` is declared to have an intent type *T*, then two constraints hold. (C1) The keys of `i` that are accessed must be a subset of *T*'s keys. It is permitted for the run-time value of variable `i` to have more keys than those listed in *T*, but they may not be accessed. It is also permitted for the run-time value of variable `i` to have fewer keys than those listed in *T*; any access to a missing key will return `null`. (C2) For every key *k* in *T*, either *k* is missing from the run-time key set of `i`, or the value mapped by *k* in the run-time value of `i` has the type mapped by *k* in *T*. This can be more concisely expressed as $\forall k \in domain(T).\texttt{i}[k] : T[k]$, where ":" indicates typing and `null` is a value of every non-primitive type.

Example The example below illustrates the declaration and use of intent types. The symbols `@A`, `@B`, and `@C` denote type qualifiers, such as @High and @Low of the noninterference type system. On the left is the type hierarchy of these type qualifiers. (C1) and (C2) are the two constraints described above.

```
                @Intent("akey" → @C) Intent i = ...
   @A           @A int e1 = i.getIntExtra("akey");  // legal
```

```
/ \          @C int e2 = i.getIntExtra("akey");  // legal
@B   @C      @B int e3 = i.getIntExtra("akey");  // violates (C2)
             i.getIntExtra("otherKey");    // violates (C1)
```

3.5.2.2 Type system rules

Figure 7 shows the typing rules for the intent type system. These rules are organized into three categories, according to their purpose. Subtyping rules define a subtyping relation for intent types, well-formedness rules define which constructions are acceptable, and typing judgment rules define the types associated with different language expressions.

Subtyping (ST) Intent type τ_1 is a subtype of intent type τ_2 if the key set of τ_2 is a subset of the key set of τ_1 and, for each key k in both τ_1 and τ_2, k is mapped to the same type.

```
@Intent("akey" → t, "anotherkey → t) Intent i1 = ...;
   @Intent("akey" →  t)) Intent i2 = ...;
@Intent("anotherkey" → t) Intent i3 = ...;
i2 = i1; // legal
i1 = i3; // illegal
```

The mapped types must be exactly the same; use of a subtyping requirement $\tau_1[k] <: \tau_2[k]$ instead of equality $\tau_1[k] = \tau_2[k]$ would lead to unsoundness in the presence of aliasing. The example below illustrates this problem. (On the left is the type qualifier hierarchy.)

```
                @C String c;
   @A           @Intent("akey" → @B) Intent i1;
   / \          @Intent("akey" → @A) Intent i2;
@B    @C        i2 = i1;    // illegal
                i2.putExtra("akey", c);
```

It would be incorrect to allow the assignment `i2 = i1` in this example, even though the assignment is valid according to standard object-oriented typing. In this case, the call to `putExtra` would store, in the object pointed by `i1`, a value of incorrect type at key `akey`. This happens because the references `i1` and `i2` are aliased to the same intent object.

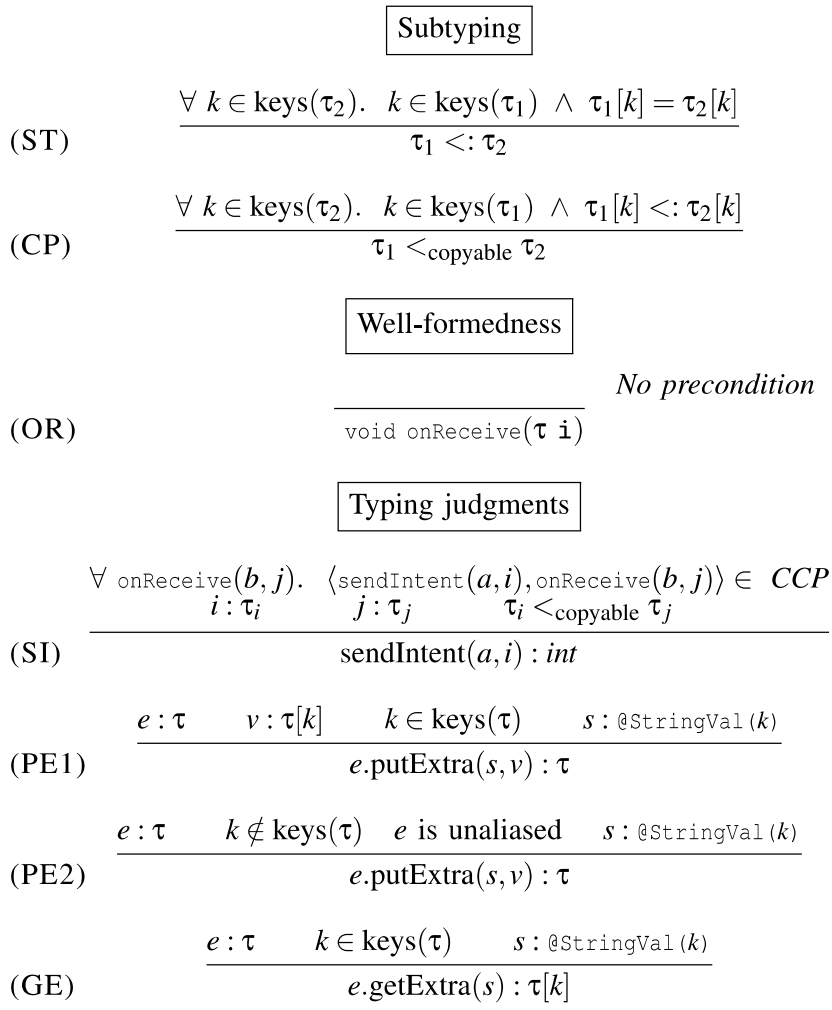

Figure 7: Type system for Android intents. Standard rules are omitted.

Copyable (CP) Copyable is a subtyping-like relationship with the weaker requirement $\tau_1[k] <: \tau_2[k]$. It may be used only when aliasing is not possible, which occurs when `onReceive` is invoked by the Android system, as explained in the (SI) rule below.

Declarations of `onReceive` (OR) A declaration of `onReceive` always type-checks. The standard Java overriding rules do not apply to declarations of `onReceive`: the intent type of the formal parameter of `onReceive` is not restricted by the type of the parameter in the overridden declaration. This is allowable because by convention `onReceive` is never called directly but rather is only called by the Android system. The type-checker prohibits direct calls to `onReceive` methods; this constraint is omitted from **Figure 7** for brevity.

Calls to `sendIntent` (SI) A `sendIntent` call can be viewed as an invocation of one or more `onReceive` methods. A `sendIntent` call type-checks if its intent argument is copyable to the formal parameter of each corresponding `onReceive` method. CCP (see Section 3.5.1) is used to determine each `onReceive` method of a `sendIntent` call. The

type comparison uses the copyable relation, not subtyping. This is sound because the Android system passes a copy of the intent argument to `onReceive`, so aliasing is not a concern.

Calls to putExtra (PE) If the receiver of a `putExtra` call might have aliases, then the argument's type must be a subtype of the type with the specified key in the map. This prevents an alias from modifying an intent in such a way that it violates the type of another alias. For example:

```
@Intent("akey" → @Low) Intent a = new Intent();
@Intent() Intent b = a;
@High String hs = ...;
b.putExtra("akey", hs);    // does not type-check
a.getExtra("akey");
```

If the receiver has no aliases, then the key is permitted to be missing from the map type.

Calls to getExtra (GE) The rule for `getExtra` is straightforward.

For both the PE and GE rules, the call (`putExtra` or `getExtra`) type-checks only if the key is a statically computable expression, according to the dataflow analysis of Section 3.4.1.2. For all 1,052 apps in the F-Droid repository, 93% of all keys could be statically computed.

3.5.2.3 Type inference

Annotations are rarely required within method bodies, because the intent type system performs flowsensitive local type inference. Consider the following example:

```
@Intent Intent i = new Intent(); // i has type @Intent()
i.putExtra "akey", h); // i now has type @Intent("akey"→@High)
i.putExtra("akey", l); // i now has type @Intent("akey"→@Low)
```

Because the receiver expression of these `putExtra` calls is an unaliased local variable, its type can be refined by adding the key–type pair from the `putExtra` call. We implemented a modular aliasing analysis that determines whether an expression is unaliased.

Figure 8 shows two cases for the `putExtra` type inference rules for intent types. For both cases, the key argument of the `putExtra` call must be a statically computable expression (Section 3.4.1.2) and the receiver expression must be unaliased. For the first case, if the intent type of the receiver expression does not have a key–type pair with the same key passed as an argument, then the intent type is refined with the new key mapping to the type of the value passed as argument. For the second case, if the intent type already has a key–type pair with the same key, then the type in this key–type pair is replaced by the type of the value passed as an argument. A further standard condition (omitted from **Figure 8** for brevity) is that the new intent type must be a subtype of the declared type.

3.5.3 Example

Recall the example of Figure 4. A noninterference type-checker would report false-positive warnings on lines 14–16 because the type system is unable to deduce that all extra data from the corresponding intent is of type `Low`. A developer can express this intended design by annotating the method `WordTranslator.getIntent` (inherited from class `Activity`):

$$\frac{e.\text{putExtra}(s,v) \quad e:\tau \quad v:\sigma \quad k \notin \text{keys}(\tau) \quad e \text{ is unaliased} \quad s:\texttt{@StringVal}(k)}{e:\tau \cup \{k \to \sigma\}}$$

$$\frac{e.\text{putExtra}(s,v) \quad e:\tau \cup \{k \to \_\} \quad v:\sigma \quad e \text{ is unaliased} \quad s:\texttt{@StringVal}(k)}{e:\tau \cup \{k \to \sigma\}}$$

Figure 8: Flow-sensitive type inference rules for intent types: the conclusion shows the type of *e* after the call to `putExtra`. Standard rules are omitted.

```
@Override
@Intent("source" → @Low, "target" → @Low, "word" → @Low)
Intent getIntent() { return super.getIntent(); }
```

The `startActivity(i)` statement on line 7 still type-checks after this change because the type-checker refines the type of `i` to `@Intent("source" → @Low, "target" → @Low, "word" → @Low)` as a result of the `putExtra` calls on lines 4–6.

The copyable typing rule enforces that the intent variable `i` in method `DictionaryMain.translateWord()` has a compatible type with the return type of `WordTranslator.getIntent()`.

By extending the noninterference type system with our intent type system and adding the correct annotations to the return type of `WordTranslator.getIntent()`, the Aard Dictionary example type-checks and the developer is assured that the program does not contain security vulnerabilities that could leak private data. Note that any developer-written annotations in the program are checked, not trusted.

4 RESULTS AND DISCUSSION

4.1 COLLABORATIVE VERIFICATION OF INFORMATION FLOW

This section describes three different evaluations of IFT. Sect. 4.1.1 describes the effectiveness of IFT in an adversarial Red Team evaluation. Sect. 4.1.2 evaluates the effectiveness and efficiency of IFT in a control team study. Sect. 4.1.3 presents a study of IFT's usability for vendors during the development of apps and for app store auditors while reviewing those apps.

4.1.1 Red Team evaluation

The sponsor of our research (DARPA) wished to evaluate IFT. To this end, they hired five development companies (in the following referred to as Red Teams) to create Android applications with and without Trojans. We had neither control over the Red

Teams nor any knowledge of the malware they were creating. While they were creating the malware, the Red Teams had access to a current version of IFT, including source code, documentation, and our own analysis of IFT's vulnerabilities. A total of

20 people worked on the Red Teams. On average they had more than 2 years of Android experience. Other than two interns, they hold BS or MS degrees and work full-time as computer security analysts. Most have been exposed to information flow theory, with the maximum experience being 6 years working with information flow.

The Red Teams created both malware and non-malware apps. The malware had to be written in

Java. The Red Teams started out by surveying real-world mobile malware. They tried to produce diverse malware, including malware that is representative of that found in the wild, novel malware that they devised, and malware specifically targeting the limitations of IFT. They had two goals: to evaluate how well IFT might work in practice, and to see how IFT could be defeated.

Overall, the Red Teams created 72 Java applications. Our sponsor provided us with the apps in five batches over an eight-month period. For each batch, we were given a few hours or days to analyze the applications with IFT. The Red Teams were given our results for the first three batches, and they used this information to create malware that was harder for IFT to find.

We received the applications in source code form. IFT does not run the applications. The applications were not obfuscated, but they were also not well-documented, and the Red Teams had no motivation to make them understandable. The user documentation was only a few sentences stating the general purpose of the app, but usually omitting significant details about the functionality

— considerably less than a typical app has in an app store. The Red Teams also had no incentive to provide code documentation or follow a specific design — code comments and design documentation

were absent, and the apps contained neither flow policies nor the information flow annotations used by IFT.

4.1.1.1 Summary of results

Of the 72 apps, 57 are malicious (see Table 3 for details).

4.1.1.2 Unjustified information flows

For 19 apps, the Android permissions in the manifest can be justified based on the purpose of the app; however, the apps leak information from one Android permission to another. For example, the app 2D Game has a malicious flow, READ_EXTERNAL_STORAGE→INTERNET. The app accesses the external storage to load photos in the game, so READ_EXTERNAL_STORAGE is justified. The app description states that the app sends high scores to a leaderboard on a server, so INTERNET is justified. The description says nothing about uploading the photos directly to the server, nor would a user expect the game to do so. Therefore, READ_EXTERNAL_STORAGE→INTERNET is a malicious flow.

An unjustified Android permission would be grounds for rejection from a high-assurance app store; however, some permissions can be easily justified. For example, one of the Red Teams used an automatic update functionality as a reason to justify the

INTERNET permission. In our experiments, we did not reject any app based on requested permissions since none of them were at odds with the app's purpose or description.

4.1.1.3 Information flows using new sources/sinks

For 17 apps, the malicious information flow is apparent only via use of the additional permissions listed in Table 2.1. For example, RSS Reader has a malicious flow of $\text{RANDOM} \rightarrow \text{VIBRATE}$. RANDOM is not an Android permission and the description of the app gives no reason to use a random number. The app is supposed to vibrate the phone when one of the user's feeds is updated, so VIBRATE is listed in the manifest file as expected. However, the app's user would not expect the app to cause random vibrations, so RANDOM→VIBRATE is malicious.

The CONDITIONAL sink detected triggers for malicious behavior in 2 apps. Countdown Timer and System Monitoring 3 triggered non-information-flow related malware after receiving SMSes with certain characters.

Other apps used time of day, random numbers, or location to trigger information-flow malware.

We found these triggers while reviewing the conditional statements.

4.1.1.4 Flows using parameterized permissions

For 11 apps, the malicious information flow is apparent only via use of parameterized permissions (Sect. 3.2.2.2). For example, in GPS 3, the location data should only flow to `maps.google.com`, but it also flows to `maps.google-cc.com`. To express this, the flow policy lists LOCATION→INTERNET("maps.google.com") but not LOCATION→INTERNET("maps.google-cc.com"). Another app, Geocaching, should only send data from specific geocaching NFC tags to the server, but it collects all NFC tags in range and sends them to the server, NFC("*")→INTERNET.

For two apps (PGP Encryption 2 and Password Saver) the leaked information is allowed to flow to the sensitive sink, but only if it is encrypted first. IFT cannot yet express this property, but Sect. 3.2.10 describes how to extend IFT to catch this sort of vulnerability.

4.1.1.5 Malware not related to information flow

The malware in 10 out of the 57 malicious applications is not related to information flow — these apps do not exhibit an unjustified information flow and implement types of attacks that are out of the scope of IFT. For example, Backup transposes digits in a phone number during backup. This is a functional correctness error, which IFT does not address. In a high-assurance app store, IFT would be used with complementary tools designed to find malware not related to information flow. The auditor's confidence that an app has no information-flow malware would let the auditor spend more time looking for other malware, such as denial of service.

4.1.1.6 Bugdoors

In 8 apps, IFT found a bugdoor (undesired, exploitable functionality) that the Red Team was unaware of. Even though the Red Team had written and/or modified the app before presenting it to us for analysis, they had not noticed these. GPS 1 passes the device ID as a way-point ID to the remote server. This allows the remote server to correlate location to specific devices and to other information collected using the device ID.

Password Saver saves unencrypted passwords in shared preferences, where they are accessible to other applications on the device.

Furthermore, 6 apps exfiltrated sensitive data to the log, which Android does not require a permission to write. It does, however, require a permission in our finer-grained permission model (see Sect. 3.2.2). Consequently, IFT reported an information flow violation.

4.1.2 Control team study

For the third and fifth batches of applications, our sponsor hired a control team to evaluate the applications and search for malware. The control team consisted of 4 PhD students with, on average, 11 years of Java experience and .5 years of Android experience — very similar to our team. The control team used both static and dynamic analysis tools, including Androguard, Eclipse, and emulators.

For the third batch of apps, the control team took an average of 9.5 hours per application to declare it malicious or benign, and their answer was right 60% of the time. Our team, using IFT plus simple tools that search for suspicious code, spent 42 minutes on average per application and correctly classified 80% of the applications.

For the fifth batch of apps, the control team took an average of 2.3 hours per application and correctly classified 63% of the apps. Our team spent 1.25 hours per app on average and correctly classified 75% of the apps.

4.1.3 Usability study

The collaborative verification model and IFT provide guarantees, but can they be used in the real world? This section presents three studies addressing this question. Sect. 4.1.3.1 measures the time to add information-flow type annotations to apps. Sect. 4.1.3.2 measures how quickly and accurately the app store auditors can approve or reject apps. Sect. 4.1.3.3 evaluates how hard it is for information-flow type system novices to learn to use IFT.

4.1.3.1 Annotation burden

In order to estimate the cost of adding information flow annotations, five members of our team annotated 11 arbitrarily chosen applications. 1 app was a malicious app written by the Red Teams and 10 apps were benign apps written by third-party developers or the Red Teams. Each annotator was given an unannotated application and a flow policy file. The annotators annotated the application until IFT issued no more warnings; if they found malware, they used a declassification and continued the task. The annotators had never seen the applications before, so the vast majority of their time was spent reverse-engineering the application.

Table 3 shows the results. On average, the annotators annotated 6 lines of code per minute, which was primarily the effort to understand the code. This compares favorably with industry-standard averages of about 20 lines of delivered code per day [13, 42, 57, 76]. (On average, the annotators annotated 20 lines of code in 3.3 minutes.) Recall that in the proposed collaborative verification model, the app's developer would annotate the code, which would be faster.

The annotated code contained on average 6 annotations per 100 lines of code. This is less than

1/4 of the annotation burden for Jif, another information-flow system for Java [5,15,92]. In our case studies, the annotator wrote an annotation in 4% of the places an annotation could have been written; the other locations were defaulted or inferred.

Table 3: Results from the annotation burden experiment

App Name	LOC	Time (min.)		De-class.	Annotations src.+sink=total		ratio
CameraTest	92	20	**.22**	1	6 + 5 = 11	**.12**	6%
Shares Pictures†	141	10	**.07**	0	12 + 0 = 12	**.09**	4%
BusinessCard	183	10	**.05**	1	9 + 0 = 9	**.05**	3%
Calculator 3	520	40	**.08**	0	7 + 0 = 7	**.01**	1%
Dynalogin	625	300	**.48**	0	66 + 0 = 66	**.11**	6%
TeaTimer	1098	295	**.27**	7	51 + 3 = 54	**.05**	3%
FourTrack	1108	120	**.11**	0	27 + 18 = 45	**.04**	3%
RingyDingy	1322	180	**.14**	2	41 + 26 = 67	**.05**	4%
VoiceNotify	1360	185	**.14**	11	68 + 44 =112	**.08**	4%
Sky	1441	240	**.17**	5	33 + 35 = 68	**.05**	3%
Pedometer	1547	165	**.11**	0	71 + 58 =129	**.08**	5%
Total	9437	1565	**.17**	26	391+189=580	**.06**	4%

Boldfaced numbers (time, annotations) are per line of code. “Declass.” is declassifications. Annotation ratio compares the number of annotations written to how many could have been written — the number of uses of types in the app’s source code. Throughout this document, lines of code (generated using David A. Wheeler’s “SLOCCount”) omit whitespace and comment lines. †Malicious applications.

Table 4: Results from the collaborative app store experiment.

App Name	Review time (min.)		Reviewed Declass.	Cond.		Accepted?
CameraTest	26	.28	1	0	0%	Accept
Shares Pictures†	5	.04	0	0	0%	Reject
BusinessCard	11	.06	1	1	14%	Accept
Calculator 3	11	.02	0	3	5%	Accept
Dynalogin	10	.02	0	10	37%	Accept
TeaTimer	50	.05	7	20	22%	Accept
FourTrack	61	.06	0	11	14%	Accept
RingyDingy	20	.02	2	11	9%	Accept
VoiceNotify	35	.03	11	73	47%	Accept
Sky	25	.02	5	19	15%	Accept
Pedometer	15	.01	0	65	57%	Accept
Total	269	.03	27	213	27%	

Boldfaced times are per line of code. All declassifications were reviewed. The Reviewed Cond. column gives the number and percentage of conditions with a sensitive source, all of which were reviewed. †Malicious applications.

The number of annotations per application is not correlated with the number of lines of code nor the number of possible annotations. Rather, the number of annotations is dependent on how, and how much, information flows through the code. When information flow is contained within procedures, type inference reduces the number of annotations required (Sect. 3.2.4.1).

4.1.3.2 Auditing burden

Another cost in the use of a static tool is the need to examine warnings to determine which ones are false positives. This cost falls on the developer who writes declassifications to suppress false positives, then again on the auditor who must review the declassifications. We wished to determine the cost of approving an app, which in addition to reviewing declassifications requires auditing the flow policy and reviewing implicit information flow.

Two graduate students acted as app store auditors. Neither one had previously used IFT or a similar framework. The auditors had never before seen the applications that they reviewed, and they did not know whether the apps were malware. The review was split into two phases: a review of the app description and flow policy, then a review of the declassifications and conditionals in the source code.

This is exactly the same workflow as an app store auditor. Table 4 summarizes the results.

The first part of the review ensures that the description of the app matches the flow policy. An auditor begins by reading the app description and writing a flow policy; then the auditor compares that to the submitted flow policy. If there is any difference, the developer must modify the description or flow policy. The flow policy review took 35% of total auditing time.

The second part of the review ensures that all declassifications and implicit information flows are valid. The auditor first reviewed the developer-written justification for each declassification. Only CameraTest had one rejected justification, which the developer rectified in a re-submission. The other justifications were accepted by the auditors. Then, the auditors investigated possible implicit information flow via conditionals (Sect. 3.2.7). Out of a total of 789 conditional statements, only 27% contained data from a sensitive source, so the auditors only reviewed those to rule out implicit information flows. For some of these conditionals, the auditor did not need to review the conditional body, because the conditional expression did not reveal anything about the content of the source. For example, 41 of the 271 conditionals with sensitive data (15%) were comparisons against null.

After the experiment, auditors mentioned that there were many unexpected flows, which ended up being necessary. Also, they wanted clear guidelines to accept or reject flow policies. We believe that both concerns will be resolved as auditors and app stores get more experience; this was their first time to audit apps.

We have not evaluated the effort of analyzing an update to an existing app, but this should be low. An update can re-use most or all of the previous flow policy specification, annotations, and justifications for declassifications.

4.1.3.3 Learnability

IFT integrates smoothly with Java and re-uses type system concepts familiar to programmers. Nonetheless, learning about information flow, or learning how to use IFT, may prove a barrier to some programmers. The programmers in the study of Sect. 4.1.3.1 were already familiar with Android and IFT. We wished to determine how difficult it is to come up to speed on IFT.

We conducted a study involving 32 third-year undergraduate students enrolled in an introductory compilers class. 60% of the students had no previous experience with Android. They received a two-hour presentation, then worked in pairs to annotate an app of 1000–1500 LOC. The apps came from the `f-droid.org` catalog; we used F-Droid because we do not have access to the source code of most apps in the Google Play Store.

The students' task was to learn Android, information flow theory, and IFT, then to reverse-engineer and to annotate the app such that IFT issues no warnings. On average the task required 15 hours. The students reported that the first annotations were the most time-consuming because they were still learning to understand IFT; after that the task was easier.

This learnability study was extremely preliminary, but it does suggest that a developer with little experience can quickly come up to speed on IFT.

4.1.4 Lessons learned

This section states a few lessons we learned during our experiments.

Generality of our analysis Our information-flow based approach turned out to be surprisingly general. IFT revealed malicious data flow of the payload as well as the injected triggers. We found, for instance, malware in applications that give wrong results based on a certain time of day or a random value. Perhaps more importantly, we were able to easily extend our system as we discovered new properties that we wished IFT to

handle — we did so over the course of our own usage and also between batches of malware analysis in the experiments.

In response to Red Team apps, we added new permissions (like RANDOM and READ_TIME), inference, intents, reflection, parameterized permissions, and more.

Effectiveness of CONDITIONAL Initially, the Red Teams used location data, time of the day, or random numbers to trigger malware. They stopped because IFT warnings made it quite easy to detect those triggers. None of the Red Teams' apps used implicit information flow maliciously — we do not know if this was because it was too hard for them or if they did not consider this attack vector.

4.1.5 Threats to validity

IFT's success in the experiments shows promise for our approach. Nonetheless, we highlight a few of the most important threats to validity in this section.

Characteristics of malware The malware we analyzed was created by five different Red Teams, each consisting of multiple engineers working full-time on the task of creating malware. The teams had previously surveyed real malware, and they created malware representative both of commercial malware that makes a profit and of advanced persistent threats who aim to steal information. Nonetheless, we have no assurance that this malware was representative of malware in the wild, either in terms of types of malware or its quality. It is also possible that IFT became tuned to the sort of malware created by those five Red Teams.

Skill of the analysts The same instrument may be more or less effective depending on who is using it. It is possible that our team was particularly skilled or lucky in classifying the apps that it analyzed — or that another team would have done a better job. An analyst needs time to come up to speed on IFT; we have found that a few weeks is sufficient for an undergraduate working part time, as confirmed by experiments (Sect. 4.1.3.3). Training only needs to occur once, and our team's unfamiliarity with the apps was a bigger impediment.

Collaborative app verification model Our model assumes that application vendors are willing to annotate their source code. We believe this is true for high-assurance app stores, but our approach may not be applicable to ordinary app stores.

4.2 SUMMARY OF MALICIOUS APPS

Table 5: Applications analyzed by IFT. All listed applications are malicious and were written by 5 independent corporate Red Teams.

	Description	LOC	Information Flow Violation	IFT
			Information flow violations involving only Android permissions	
1	Adventure Game	17,896	READ_EXTERNAL_STORAGE!WRITE_EXTERNAL_STORAGE	✓
2	Note Taker	3,251	CAPTURE_AUDIO_OUTPUT!INTERNET	✓
3	SMS Pager	1,834	READ_SMS!INTERNET	✓
4	Battery Indicator	4,214	READ_EXTERNAL_STORAGE!INTERNET	✓
5	Block SMS	2,087	RECEIVE_SMS!INTERNET	✓
6	Fortune	2,998	READ_PHONE_STATE!INTERNET	✓
7	WiFi Finder	852	ACCESS_FINE_LOCATION!INTERNET	✓
8	Replacement launcher	1,069	READ_PHONE_STATE!WRITE_EXTERNAL_STORAGE	✓
9	2D Game	33,017	READ_EXTERNAL_STORAGE!INTERNET	✓
10	Displays source code	242	READ_PHONE_STATE!INTERNET	✓
11	System Monitoring 2	9,530	ACCESS_FINE_LOCATION!WRITE_EXTERNAL_STORAGE	✓
12	SMS Encryption	27,764	READ_SMS!SEND_SMS	✓
13	Bible	19,775	INTERNET!WRITE_EXTERNAL_STORAGE	✓
14	GPS 1	720	READ_PHONE_STATE!INTERNET	✓
15	GPS Logger	6,907	ACCESS_FINE_LOCATION!INTERNET	✓
16	Shares Pictures	135	READ_EXTERNAL_STORAGE!INTERNET	✓
17	Cat Pictures	639	READ_EXTERNAL_STORAGE!INTERNET	✓
18	SMS Messenger	1,210	READ_SMS!WRITE_SMS	✓
19	Running Log	1,333	READ_PHONE_STATE!NFC	✓
			Information flow violations involving IFT's additional permissions	
20	Countdown Timer	1,065	RECEIVE_SMS!CONDITIONAL	✓
21	Cookbook	2,542	LITERAL!WRITE_CONTACTS	✓
22	SMS Notification	9,678	READ_SMS!WRITE_LOGS	✓
23	Calculator 2	640	USER_INPUT!FILESYSTEM	✓
24	SMS Backup	293	READ_EXTERNAL_STORAGE!WRITE_LOG	✓
25	Password Protects Apps	11,743	RANDOM!MODIFY_PHONE_STATE	✓
26	System Monitoring 1	9,402	LITERAL!WRITE_SETTINGS	✓
27	Calculator 1	510	RANDOM!DISPLAY	✓
28	RSS Reader	3,503	RANDOM!VIBRATE	✓
29	Text to Morse code	263	USER_INPUT!FILESYSTEM	✓
30	Shares Location	248	ACCESS_FINE_LOCATION!PROCESS_BUILDER	✓
31	Calculator 4	482	RANDOM!DISPLAY	✓
32	Device Admin 1	1,474	ACCESS_FINE_LOCATION!INTENT	✓
33	Device Admin 2	1,700	FILESYSTEM!INTERNET	✓
34	DropBox Uploader	5,902	DISPLAY!INTERNET	✓
35	System Monitoring 3	3,334	RECEIVE_SMS!CONDITIONAL	✓
36	Phone silencer	1,415	LITERAL!MODIFY_PHONE_STATE	✓
			Information flow violations involving parameterized permissions	
37	Screen Saver 1	147	LITERAL("")!WRITE_EXTERNAL_STORAGE	✓
38	GPS 3	1,512	LOCATION!INTERNET("maps.google-cc.com")	✓
39	Geocaching	27,892	NFC("*")!INTERNET	✓
40	Instant Messenger	1,253	LITERAL("0xFFFF")!INTERNET	✓
41	App Backup	2,010	LITERAL!WRITE_EXTERNAL_STORAGE("*")	✓
42	Mapping	5,587	LOCATION!INTERNET("mapxplore.com")	✓
43	SIP VoIP Phone	1,480	USER_INPUT !USE_SIP("2233520413@sip2sip.info")	✓
44	Word Game	1,191	LITERAL !SEND_SMS("12025551212")	✓
45	PGP Encryption 1	9,904	USER_INPUT("EditText.passPhrase")!EMAIL	✓
46	PGP Encryption 2	9,945	USER_INPUT("EditText.message")!EMAIL	✓
47	Password Saver	508	USER_INPUT("EditText.createPassword")!SHARED_PREFERENCES	✓
			Malware not related to information flow	
48	Podcast Player	1,711	none — Battery DoS	✓
49	Screen Saver 2	419	none — Battery DoS	✓
50	To Do List	5,123	none — Battery DoS	✓
51	Sudoku	1,505	none — Battery DoS	✓
52	Expense reports	2,293	none — Performance DoS	✓
53	Automatic SMS replies	33,296	none — Performance DoS	✓

54	Screen Saver 3	457	none — Performance DoS	✓
55	Backup	2,554	none — Data corruption	✓
56	SMS Reminders	2,917	none — Data corruption	✓
57	Game 3	1,211	none — Clickjacking28	✓

✓ The malicious flows or permissions in these apps were found using IFT.

* These malicious flows will be caught by IFT after future work is complete.

4.3 STATIC ANALYSIS OF IMPLICIT CONTROL FLOW

Our implementation works on Java code: it does not analyze native calls. For efficiency, it relies on trusted annotations for system libraries. These are standard limitations of a static analysis. Section 3.5.1 notes other limitations regarding the estimation of component communication patterns.

Modulo these limitations, our analysis is sound. That is, if a program type-checks, then the type of any expression is a sound estimate of its possible run-time values.

For reflection, this means that the value for a `Class` or `Method` expression is contained within the set of possible values in its type, and likewise for array lengths.

For intents, this means that if an expression has a type with an intent key–type pair, then at run time the expression's value is an intent whose extra data maps the key to a value of that type, or the key does not appear in the map.

Equally importantly, the resolution preserves any soundness property for a downstream analysis. If the downstream analysis is sound when using the conservative library annotations, then it remains sound when using more precise summaries supplied by the reflection and intent resolvers.

It is possible to state formal type-correctness, progress, and preservation theorems for our type systems. The theorems are standard and their proofs would be straightforward.

Table 6 Selected subject apps from the F-Droid repository. The number of reflective invocatons is given for `MethodS` and `Constructors`, and intent uses count the number of `putExtra` and `getExtra` calls. The last three columns show the annotation overhead for the technique IFT+INT+RR. The column IFT shows the number of `@Source` and `@Sink` information flow annotations. The column refl shows

the number of `@MethodVal` and `@ClassBound` annotations (no `@ClassVal` annotations were required). The column int shows the number of `@Intent` annotations.

App	LOC	Reflection		Intent uses		# of annotations		
		meth	cons	put	get	IFC	refl	int
AbstractArt	4,488	1	0	1	1	317	0	1
arXiv	3,643	14	0	70	17	130	0	13
Bluez IME	4,523	4	2	124	42	285	0	16
ComicsReader	6.612	6	0	1	2	381	1	6
MultiPicture	7,496	1	0	17	12	511	0	17
PrimitiveFTP	4,026	2	0	1	1	321	0	1
RemoteKeyboard	5,723	1	0	3	4	580	0	4
SuperGenPass	2,125	1	0	15	14	181	0	8
VimTouch	8,881	1	0	7	6	2,424	2	7
VLCRemote	5,097	1	0	12	21	453	0	22
Total	52,614	32	2	251	120	5,583	3	95

4.4 IMPROVING A DOWNSTREAM ANALYSIS

We evaluated our work in two ways. First, this section reports how much our reflection and intent analyses improve the precision of a downstream analysis, which is their entire purpose. Second, Section 4.5 measures how well our type inference rules reduce the programmer annotation burden.

4.4.1 Subject programs and downstream analysis

We used open-source apps from the F-Droid repository [26] to evaluate our approach. F-Droid contains 1,052 apps that have an average size of 9,237 LOC[4] and do not use third-party libraries.

415 out of 1,052 F-Droid apps (39%) use reflection, and each app that uses reflection has on average 11 reflective method or constructor invocations. 726 out of 1,052 F-Droid apps (69%) use intents with extra data, and each app that uses intents with extra data has on average 24 calls to `putExtra` or `getExtra`. 254 out of 1,052 F-Droid apps (24%) use both reflective calls *and* intents with extra data.

These numbers support our motivation to pursue static analysis of reflection and intents.

We aimed to select subject apps of typical complexity. We excluded excessively simple apps: those with less than 2,000 LOC or that did not have at least one call to `putExtra`, `getExtra`, and `Method.invoke`. We also excluded excessively complex apps: those with more than 15,000 LOC or that used more than five Android permissions, which is the average number of permissions used by an F-Droid app.

[4] Non-comment, non-blank lines of code, as reported by David A. Wheeler's SLOCCount. See `http://www.dwheeler.com/sloccount/`.

Overall, 40 apps satisfied our requirements, and we randomly sampled 10 apps, which are listed in Table 6. Each of the 10 apps contains on average 5,261 LOC, 3 reflective method or constructor invocations[5], and 37 calls to `putExtra` or `getExtra`.

Our evaluation uses three downstream analyses. Sections 4.4.2-4.4.3 discuss the Information Flow Checker (IFT); Section 4.4.4 briefly discusses the other two case studies. IFT is a type system and corresponding type-checker that prevents unintended leakage of sensitive data from an application [23]. Given a program and an information-flow policy (a high-level specification of information flow, expressed as source–sink pairs), IFT guarantees that no other information flows occur in the program. IFT is sound: it issues a warning if the information flow type of any variable or expression does not appear in the information-flow policy. IFT is also conservative: if it issues a warning, then the program might or might not misbehave at run time.

We evaluated the effectiveness of our techniques by studying the following two research questions.

4.4.2 How much do our reflection and intent analyses improve the precision of IFT?

We measured the precision and recall of IFT's static estimate of possible information flows. To compute precision and recall, we manually determined the ground truth: the actual number of flows that could occur at run time in an app.[6] Precision is the number of ground-truth flows, divided by the total number of flows reported by the analysis. Recall is the number of real flows reported by the analysis, divided by the total number of ground-truth flows. We confirmed that IFT has 100% recall both with and without the reflection and intent analyses, i.e., IFT is sound and misses no real flows.

To evaluate this research question, we compared the precision of the following techniques.

IFT-unsound makes optimistic assumptions about every reflective and intent-related call. Its recall is only 95% — it unsoundly misses 5% of the information flows in the apps, which makes it unacceptable for use in the security domain. Its precision was 100%, for this set of apps.

IFT treats reflection and intents conservatively. Data in an intent may be from any source and may flow to any sink. Data used as an argument to a reflective invocation may flow to any sink, and data returned from a reflective invocation may be from any source. In the absence of reflection and intents, IFT is an effective analysis with high precision, as shown by IFT-unsound. However, for our subject programs, which use reflection and intents, IFT's precision is just 0.24%.

IFT+RR augments IFT with reflection resolution and can therefore treat data that is used in reflection precisely when the reflection can be resolved. Data in intents, however, is treated conservatively. Since all apps send intents, which may trigger the use of any permissions, reflection resolution alone does not help; the average precision

[5] This is smaller than the F-Droid average: most uses of reflection in F-Droid appear in a few huge apps (>500 kLoC) that contain hundreds of reflective calls.

[6] This enormous manual effort is the reason we did not run the experiments on all 1,052 F-Droid apps. It would be easy to run our analyses on all the apps, but doing so would not indicate whether our analyses were useful.

remains 0.24%. In a (non-Android) program that does not use intents, IFT+RR would outperform IFT.

IFT+INT augments IFT with intent analysis. It reports precise information flows for method calls involving intents. Differently from intent invocations, reflective calls are only allowed to use permissions listed in the app's manifest. Therefore, data passed to a reflective invocation is treated as flowing to any sink the app may access. Similarly, data returned from a reflective invocation is treated as if it could have come from all sources listed in the manifest. However, since Epicc generates CCP and unsoundly assumes that reflective calls do not invoke `sendIntent` methods, IFT+INT must issue a warning any time a method is reflectively invoked. For each such warning, the developer must manually verify that the reflective method does not invoke `sendIntent`. The average precision is 53%.

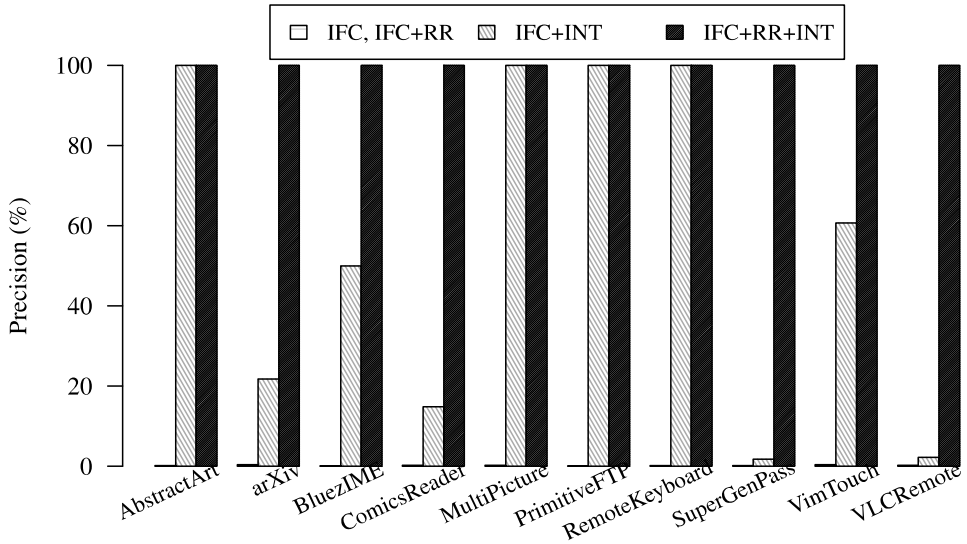

Figure 9 Comparison of precision among techniques.

IFT+INT+RR augments IFT with both reflection resolution and intent analysis. When reflection resolution cannot resolve a method or when it resolves a method to `sendIntent`, it still issues a warning. The precision is 100% for each of the 10 randomly-chosen apps, but might be smaller for other apps.

Figure 9 plots the precision for the sound techniques.[7] Being the most basic technique, IFT has the worst precision among all approaches. At the other extreme, IFT+INT+RR has the highest precision for all cases. This occurs because this technique provides custom support for both reflective calls and intents. Such high precision is obtained at the expense of adding annotations in the code. Section 4.4.3 discusses the overhead associated with the annotation process.

[7] All sound techniques achieve 100% recall.

IFT+INT has perfect precision for `AbstractArt`, `MultiPicture`, `PrimitiveFTP`, and `RemoteKeyboard`, because these apps use reflection for control flow but not data flow — data returned from reflective calls is not sent to a sensitive sink and no sensitive information is passed as an argument to a reflective method call. For the other 6 apps, IFT+INT is more precise than IFT, but still reports flows that cannot happen at run time. For these apps, the reflection resolver is needed to reach 100% precision as reported by IFT+INT+RR. The results confirm that both techniques, reflection resolution and intent analysis, are necessary and that they are complementary and synergistic.

The 10 apps in Figure 9 use both reflection and intents with extras, like 24% of all apps in the F-Droid repository. For apps that use just one of the features, IFT+RR or IFT+INT would achieve the same precision as IFT+INT+RR.

All uses of reflection could be resolved except for one in the `RemoteKeyboard` app. For that case, the reflection resolver could determine the name of the invoked method (`createShell`) and the number of parameters (0), but the class name is obtained from preferences that the user can edit at run time. However, this method is not `sendIntent`, its returned object is not sent to any sink, and it takes no parameters; therefore, treating that call conservatively did not decrease the precision of IFT+INT+RR.

We attempted to compare our approach with IccTA [50]. IccTA crashed when run on 1 of the 10 apps. For the other 9 apps, IccTA outputted some static analysis data, but no data regarding information flows. We contacted the IccTA authors about these issues but didn't hear back from them.

4.4.3 What is the annotation overhead for programmers?

Developers must write source code annotations in order to use our analyses. This is not extra work, since the alternative would be to spend time reviewing false-positive warnings.

Table 6 shows the annotations required to type-check each app. Less than 2% as many annotations are required due to reflection and intents, compared to annotations related to information flow (the downstream analysis). If the programmer omits an annotation, or writes one that is inconsistent with the source code or with other annotations, then the analyses issue user-friendly warnings that pinpoint and explain the type inconsistency. The average time to add each annotation related to our analyses was roughly one minute, for a member of our team.[8] Thus, the annotation overhead is small in absolute and relative terms, especially considering the significant improvements in precision due to reflection and intent analysis.

Part of the need for annotations is because the downstream analysis is a modular analysis — a type-checker that verifies programmer-written types. If the downstream analysis were a whole-program analysis such as pointer analysis, type inference, or abstract interpretation, these would not be necessary. Other annotations are needed to express facts that no static analysis can infer; in these cases, human intervention is unavoidable.

[8] A developer who is familiar with the subject programs might take less time. The developer would need to learn to use IFT, but we have found that doing so is straightforward for someone who understands information flow.

4.4.4 Precision improvements for other downstream analyses

We demonstrated the generality of our approach by integrating our reflection and intent analyses with two other downstream analyses. The Nullness Checker [67] verifies the absence of null pointer dereferences: if the Nullness Checker issues no errors for a given program, then that program is guaranteed to not throw a `NullPointerException` at runtime. The Interning Checker [67] verifies equality tests: if the Interning Checker issues no errors for a given program, then all reference equality tests (i.e., ==) are over canonicalized data, and thus are consistent with `.equals()`.

These analyses suffer false positives due to reflection and intents. Consider the Nullness Checker.

Its library annotations must mark the return type of `Method.invoke` as `@Nullable`, for soundness. The reflection analysis can determine that some calls to `invoke` return a non-null value, and thus it eliminates false positives in the nullness analysis.

Reflection resolution improved the precision of the Nullness Checker for 3 of the 10 apps. There were no reference equality tests over values returned by a reflective method invocation, and therefore reflection resolution did not improve the precision of the Interning Checker for these 10 apps.

The intent analysis improved the precision of the Interning Checker for 2 of the 10 apps. The intent analysis does not improve the precision of the Nullness Checker for any app, because `getExtra` can return `null` if a key does not exist in an intent map. The intent type system does not guarantee the existence of a key in an intent map — only that if it exists, it has a certain type.

4.5 EVALUATION OF TYPE INFERENCE

As shown in Section 4.3, programmers have to write very few annotations to aid the reflection and intent analysis. This section explains why, by evaluating our type inference rules.

4.5.1 Reflection resolution

In addition to the 10 subject apps of Section 4.4, we arbitrarily selected 25 apps from F-Droid that use reflection. Using the entire set of 35 apps, we evaluated the reflection resolution by answering the following three research questions.

4.5.1.1 How is reflection used in practice?

The 35 apps contain 142 invocations of reflective methods or constructors. 81% are used to provide backward compatibility, 6% access a non-public API, and 13% are for other use cases.

4.5.1.2 How often can reflection be resolved at compile time?

Our reflection resolution resolved 93% of instances of reflective method or constructor invocations. It failed on the other 7% because the reflectively invoked method or constructor cannot be determined statically by any analysis. As an example, the `RemoteKeyboard` app uses reflection for extensibility and duck typing: the user can configure the class name for a shell implementation, and the app reflectively invokes a factory method on this class. Moreover, these shell implementations do not have a

common interface that defines the factory method, rendering static reflection resolution impracticable.

4.5.1.3 How effective is type inference for reflection resolution?

To enable modular reflection resolution, a developer may have to write type annotations in a program. We evaluated the effectiveness of our type inference (see Section 3.4.1.2) that reduces the annotation burden. Specifically, we determined how many instances of reflection can be resolved without any developer-written annotation and whether the remaining instances require stronger inference or developer-written annotations.

For 52% of reflective invocations, our intra-procedural type inference enabled fully automated reflection resolution. This means that our type inference determined the exact method that is reflectively invoked without requiring a single annotation.

For 41% of reflective invocations, our inter-procedural, intra-class type inference determined the exact method that is reflectively invoked. A common example is the initialization of a private field of type `Class` or `Method`. These fields are only assigned once but are initialized within a method that provides exception handling. Another example is the use of a helper method that manipulates Strings and returns an object of type `Method` that is used within the class.

We also implemented an inter-class inference, but it did not improve the results for the selected apps, beyond the intra-class analysis results.

The other 7% of reflection invocations cannot be resolved by any static analysis (for an example, see Section 4.5.1.2). Code inspection and developer intervention are required in those cases.

Table 6 gives the number of developer-written annotations that were required. Recall that all annotations in an app are checked, not trusted. Thus, use of developer-supplied annotations does not compromise the soundness of our approach.

4.5.1.4 Bug detection

Our reflection resolver revealed a bug in the `arXiv` app. The reflection resolver reported an unresolvable method even though it precisely inferred the class name, method name, and the number of parameters. The bug was a misspelled method name, and it prevented a menu from being updated. The developer confirmed the bug.

4.5.2 Intent type inference

Section 3.5.2.3 introduced rules to refine the type of an intent, which reduce the number of developer written annotations required in a program. This section evaluates how effective they are in practice. We only implemented type refinement for sent intents. A limitation of our implementation is that declarations of `onReceive` methods must have a precise intent type, so `sendIntent` calls can be typechecked against these declarations. Therefore, we evaluated type refinement of sent intents (68% of all intents). We defer inferring intent types on declarations of `onReceive` methods to future work. We considered only intents with extras (51% of all sent intents), as an empty intent requires no developer-written annotation.

To measure the effectiveness of the intent type inference (Section 3.5.2.3), we used a similar approach as when measuring the reflection resolution type inference: we

determined the number of sent intents with extras that required no annotations and compared it with the overall number of sent intents with extras.

For 67% of the cases, our intra-procedural inference determined that the sent intent had no aliases and precisely inferred the type of the sent intent. For those cases, developer-written annotations are not necessary.

For 21% of the cases, our inter-procedural inference correctly infers the type of the sent intent.

For 12% of the cases, the sent intent was stored in a field. Our alias analysis (Section 3.5.2.3) treated such intents as possibly-aliased, so the intent type cannot be refined using the `putExtra` rule.

The 10 apps require a total of 7 developer-written annotations for sent intents with extras. Without intent type inference, the apps would have needed an additional 52 developer-written annotations in order to type-check[9]. This result shows that intent type inference greatly reduces the annotation burden.

4.6 RELATED WORK

4.6.1 Information flow

Information flow tracking has been investigated for several languages and paradigms [29,38,51,69,88]. These approaches are largely complementary to our work as they are theoretical or do not employ type systems to achieve static guarantees of information flow properties. Besides statically verifying properties, several approaches for enforcing information flow properties have been proposed, such as refactoring [75], dynamic analysis [56], or encoding as safety properties [61,79]. Milanova and Huang [58] recently presented a system that combines information flow with reference immutability to improve precision. Yet, the system has not been applied in a security context. Engelhardt et al. [22] discuss handling intransitive information-flow policies; IFT makes transitive flows explicit. Sun et al. [77] discusses modular inference for information flow; IFT does inference within method bodies.

In the domain of information flow tracking for Java programs, the closest related work is Jif (Java information flow) [59,60,72]. Jif uses an incompatible extension of the Java programming language and its own compiler to express and check information flow properties of a program. In contrast, IFT uses standard Java annotations and the code can be compiled with the standard Java compiler. Furthermore, IFT achieves its effects with a simpler, easier-to-use type system. While Jif focuses on the expressiveness and flexibility of the type system and trust model, IFT aims at practicality and scalability to be applicable on large real-world Android applications. IFT has better support for defaults, inference, reflection, intents, libraries, separate compilation, and many other language features. Jif has not been evaluated in an adversarial challenge exercise comparable to our experiments using IFT.

In the context of implicit information flow, the classic approach is to taint all computations within the dynamic scope of a conditional statement. Suggested by Denning and Denning [17] and formulated as a type system by Volpano et al. [83], this

[9] 88% 6= 52/(7+52) because some developer-written annotations solve multiple cases where intent type inference does not succeed.

approach causes over-tainting and suffers from taint explosion. Kang et al. [43] investigated the problem of under-tainting in benign applications. They found that under-tainting usually occurs at only a few locations and proposed an analysis to identify and taint such additional targets. However, malicious applications are out of scope.

WebSSARI [40] focuses on web applications written in PHP and aims at preventing vulnerabilities such as Cross-Site Scripting or SQL Injection. In this context, static analysis is applied to reveal existing weaknesses and to insert run-time checks. In contrast, IFT statically verifies information flow properties for Android applications.

4.6.2 Android malware

Many Android apps are overprivileged, i.e., they are granted more permissions than they use [8,27,82]. These studies also provided a mapping of API calls to required permissions. IFT utilizes those existing mappings and enhances the Android permission system by adding finer-grained sources and sinks for sensitive APIs. Chin et al. [14] described a weakness of Android Intents: implicitly sent intents can be intercepted by malicious applications. IFT analyzes communication through intents to detect such attacks.

The Google Play Store runs Bouncer to detect and reject malicious applications. Unfortunately, Bouncer can be circumvented [44,68], which motivates our work. Ongtang et al. [65] suggest an application-centric security model to strengthen Android's security.

Woodpecker [34] uses static analysis to detect capability leaks. ComDroid [14] uses static analysis to locate Intent-related vulnerabilities. Several systems have been proposed to detect the leakage of personal data (e.g., [31,55]). In this context, PiOS [19] detects privacy leaks in iOS applications by constructing a control flow graph from compiled code and performing data flow analysis. FlowDroid [6] is a static taint analysis tool for Android apps that that has not been used to find malware. FlowDroid propagates sources and sinks found using SuSi [71], which uses machine learning to classify and categorize Android library methods as source and sinks. Unlike those existing approaches, IFT uses a finer-grained model for sources and sinks, operates on the source code, and is not limited to explicit information flow. RiskRanker [35] and DroidRanger [94] combine multiple analyses in an attempt to detect likely malware.

Beyond detection, dynamic enforcement tools have been proposed to monitor the execution of an application at run time and intervene, if necessary, to ensure safe behavior [21,39,89,90]. These techniques are non-portable or suffer high overheads. Another disadvantage of a dynamic analysis is that it may cause an app to fail at run time. By contrast, a static analysis such as IFT gives a guarantee ahead of time, with no run-time overhead, no special runtime environment, and no risk of failures in the field.

4.6.3 Collaborative model

A similar collaborative verification model has been proposed in prior work on the verification of browser extensions. For example, Guha et al. [37] describe a model in which browser extension developers specify a policy; as in our approach, the program's adherence to the policy is statically verified, and the reasonableness of the policy is manually verified by an auditor. IFT applies the collaborate verification approach to Android, with a significantly simpler policy language, easing the auditor's burden of verifying the reasonableness of the policy.

Similarly, Lerner et al. [48, 49] extend JavaScript with a type system to statically verify that extensions do not violate the browser's private browsing mode; their approach requires developers to write annotations only where code might violate private browsing expectations. It also requires a skilled auditor to manually verify declassifications. IFT poses a lower annotation and audit burden and supports a broader range of information flow guarantees.

Our collaborative verification model requires a trained auditor to ensure an app's description matches the app's flow policy. Other work has used crowd-sourcing [1], natural language processing [66], or clustering [33] to verify that an app's description matches an app's functionality. The functionality is modeled by private information accessed, permissions requested, or APIs used. These techniques could be modified to compare a flow policy to an app's description, thereby reducing the auditors effort.

4.6.4 Reflection

The most common approach for improving precision of a static analysis in the presence of reflection is profiling from an observed set of executions, assuming that the observed program exercises all possible behaviors. Livshits [54] requires user annotations or dynamic information from casts to estimate reflection targets as part of static call graph construction. Tatsubori [78] earlier built a system with similar qualities. TamiFlex [11] performs unsound dynamic analysis of reflection and dynamic class loading. It replaces uses of reflection by standard method calls, and supplies the modified call graphs to existing static-analysis tools. In other words, an unsound analysis can be built on top of TamiFlex, just as a sound analysis can be built on top our work. An example is that Averroes [2] can use TamiFlex when building call graphs, to unsoundly improve precision over its conservative defaults. All of these approaches that use dynamic information are unsound. By contrast, our approach is sound: it makes conservative assumptions about any occurrence of reflection that it cannot handle.

In some special cases, reflection can be resolved based on assumptions about the run-time execution context. For example, Zhang's GUI error detection tool [91] builds reflection-aware call graphs for Android applications, enabling it to find more GUI errors than without. However, it only handles a particular scenario — it converts reflective calls into explicit constructor invocations based on the contents of configuration files at compile time. This approach is sound if the same configuration files will be installed at run time as at analysis time. This is the same assumption made by Epicc [64] to handle inter-component communication, which our system uses.

A few static analyses partially handle reflection. Javari [81] introduces a new API to invoke reflection that does a single dynamic check of the method signature rather than of the object. Programs using that API can be soundly type-checked. Our approach could eliminate that special API and the run-time check. Li et al. [52] developed an unsound self-inferencing reflection resolution to improve the precision of a pointer analysis for Java programs. They additionally analyzed how reflection is used in open-source Java applications. In contrast, our approach is sound and our evaluation focuses on the use of reflection in Android apps.

4.6.5 Android

We evaluated our reflection and intent analyses in the context of detecting and preventing malicious behavior in mobile apps [14,20,21,30,31,34,35,39,55,89,90,94]. We discuss some closely related work.

SCanDroid [30] applies data flow analysis to check security properties in Android apps. It analyzes intra-component and inter-component information flows for vulnerabilities. The analysis cannot handle interactions between apps and provides limited support to handle intent extras, making no distinction between the flows of permissions that result from the entries of an intent. Several other techniques came after it [7,14,20,31,34,35,55,94], improving precision and recall of reported warnings. However, to the best of our knowledge, no later technique has focused on handling the important aspect of data encapsulation in intents. Our technique is complementary to push-button static analysis techniques such as SCanDroid: our analysis requires a small number of annotations from the developer but requires less examination of false positives and provides stronger guarantees. It preserves soundness, achieves good precision, and remains easy to use.

FlowDroid [7] is a technique that performs taint analysis on Android apps with the goal of finding security vulnerabilities. FlowDroid does not support Android's implicit intents nor reflection. In experiments, the tool achieved 83% precision and 93% recall for apps containing different types of vulnerabilities.

Our implementation currently relies on Epicc [64] to approximate the set of component pairs that actually communicate. See Section 3.5.1 for a discussion.

Our implementation has been publicly available since December 12, 2013. In forthcoming work, IccTA [50] adopts a similar approach that performs static taint analysis in the presence of intercomponent communication. IccTA's reflection resolution is much more limited than ours: it only processes string constants. Although IccTA is applied to taint analysis, IccTA is neither sound nor complete; by contrast to our work, it provides no security guarantees to its user and is not applicable in the context of high-assurance app stores [23]. Even if the analysis flaws were addressed, IccTA would remain vulnerable because its taint model uses an insufficient set of sensitive sources and sinks. Another difference is the evaluation: we measured the precision and recall of our information-flow analysis on real Android apps and achieved 100% precision and recall, but IccTA was evaluated on 22 examples hand-crafted by its authors, where it achieved 96% precision and recall.

4.6.6 Other

Xiao et al. [88] proposed a semi-automatic approach to analyze TouchDevelop mobile app scripts for privacy. Their workflow is similar to ours: users annotate APIs and code, and the analyzer uses a dataflow analysis to check conformance of inferred flows against a specification of the app. However, their static analysis does not handle implicit control flows.

Google's Android NDK [3] allows parts of an app to be implemented using native-code languages such as C and C++. Our toolset does no analysis of native code: summaries for native methods are trusted. The Checker Framework, on which our implementation is built, treats unannotated methods conservatively.

Our work has some similarities to call graph construction in object-oriented programs [47,80]. Dynamic dispatching can be viewed as an implicit control flow

mechanism, much as Java reflection and Android intents can. Most call graph construction algorithms do whole-program pointer analysis. Our approach is modular but relies on user annotations. A whole-program type inference or pointer analysis could eliminate the need for programmers to write annotations.

4.7 CHECKER FRAMEWORK IMPROVEMENTS

Over the course of the project we fixed over 210 issues in the Checker Framework's bug tracker. In addition, we added significant new functionality to the Checker Framework. The bug fixes and enhancements were important to the SPARTA project, and they also aid other users of the Checker Framework. This section highlights a few of these improvements.

The Checker Framework's verification methodology — pluggable typechecking — was so compelling that Oracle supported an extension to the Java language to support it. This feature is called type annotations and is also known by its codename JSR 308. Type annotations are an official part of the Java 8 language and are supported by every Java implementation. The Checker Framework uses Java's type annotation capability when present, though the Checker Framework is also backward-compatible

with earlier versions of the Java language.

4.8 DATAFLOW FRAMEWORK

The dataflow framework enables more accurate analysis of source code. (Despite their similar names, the dataflow framework is independent of the (Information) Flow Checker of chapter 2.) In Java code, a given operation may be permitted or forbidden depending on previous actions; for example, a file should be read only if it has been previously opened.

The primary purpose of the Dataflow Framework is to enable flow-sensitive type refinement in the Checker Framework. In other words, a variable can have a different type on different lines of code, depending on reassignments, tests, and method calls. As an example, a variable `x` might be possibly null, but after a test `if (x = null)` or after a reassignment `x = methodThatReturnsNonNull()`, the variable is known to be non-null. flow-sensitive type refinement reduce the programmer's burden of annotating a program.

The Dataflow Framework's result is an abstract value for each expression (an estimate of the expression's run-time value) and a store at each program point. A store maps variables and other distinguished expressions to abstract values. An expression's abstract value, as computed by the dataflow framework,

As a pre-pass, the Dataflow Framework transforms an input AST into a control flow graph consisting of basic blocks made up of nodes representing single operations. To produce its output, the Checker

Framework performs iterative data flow analysis over the control flow graph. The effect of a single node on the dataflow store is represented by a transfer function, which takes an input store and a node and produces an output store. Once the analysis reaches a fixed point, the result can be accessed by client code.

In the Checker Framework, the abstract values to be computed are annotated types. An individual checker can customize its analysis by extending the abstract value class and by overriding the behavior of the transfer function for particular node types.

The dataflow framework contains distinct components for building a control flow graph (CFG) and performing the fixed-point dataflow analysis itself. It contains a semi-

declarative mechanism for specifying the transfer functions and a fixpoint mechanism for terminating the iteration. It handles multiple type hierarchies simultaneously and is able to transfer information between the CFG and the dataflow analysis. This framework is useful for other analyses beyond type-checking. For instance, Google's Error Prone tool (`http://errorprone.info/`) discarded its dataflow analysis and now uses our dataflow framework instead. As a result, the new version of Error Prone discovered many new flaws in Google's codebase.

The Dataflow Framework was designed with several goals in mind. First, to encourage other uses of the framework, it is written as a separate package that can be built and used with no dependence on the Checker Framework. Second, the framework is currently intended to support analysis but not transformation, so it provides information that can be used by a type checker or an IDE, but it does not support optimization. Third, the framework aims to minimize the burden on developers who build on top of it. In particular, the hierarchy of analysis classes is designed to reduce the effort required to implement a new flow-sensitive type checker in the Checker Framework. The Dataflow User's Guide (`http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#dataflow`) explains how to customize dataflow to add checker-specific enhancements.

4.9 NEW TYPE SYSTEMS

In addition to the type systems described in chapters 2 and 3, we implemented several other new type systems. These are publicly distributed with the Checker Framework, and each one is fully described by a chapter of the Checker Framework Manual.

Initialization Checker Code can suffer a `NullPointerException` when using a non-null field, if the code uses the field during initialization. That is because every object's fields start out as null. By the time the constructor finishes executing, the non-null fields have been set to a different value. The Initialzation Checker determines which fields are initilaized. This information is used by the Nullness Checker, which warns whenever an uninitialized field may be accessed.

Lock Checker The Lock Checker prevents certain concurrency errors by enforcing a locking discipline. A locking discipline indicates which locks must be held when a given operation occurs. A programmer expresses the locking discipline via the type qualifier `@GuardedBy("lockexpr")`. This indicates that the expression's value may be dereferenced only if the given lock is held. Our semantics and analysis [24,25] are unique in that they analyze program values, just as the run time system does, rather than an unsound approximation such as field names.

Format String Checker The Format String Checker [86,87] prevents use of incorrect format strings in format methods such as `System.out.printf` and `String.format`. The Format String Checker warns programmers if they write an invalid format string, and it warns if the other arguments are not consistent with the format string (in number of arguments or in their types).

Internationalization Format String Checker The Internationalization Format String Checker, or I18n Format String Checker, prevents use of incorrect i18n format strings. If the I18n Format String Checker issues no warnings or errors, then `java.text.MessageFormat.format` will raise no error at run time.

Internationalization Checker The Internationalization Checker, or I18n Checker, verifies that code is properly internationalized. Internationalization is the process of designing software so that it can be adapted to different languages and locales without needing to change the code (only resource files need to be changed). Localization is the process of adapting internationalized software to specific languages and locales.

The checker focuses on one aspect of internationalization: user-visible strings should be presented in the user's own language, such as English, French, or German. This is achieved by looking up keys in a localization resource, which maps keys to user-visible strings. For instance, one version of a resource might map "`CANCEL_STRING`" to "`Cancel`", and another version of the same resource might map "`CANCEL_STRING`" to "`Abbrechen`".

The Internationalization Checker verifies these two properties:

1. Any user-visible text should be obtained from a localization resource. For example, `String` literals should not be output to the user.
2. When looking up keys in a localization resource, the key should exist in that resource. This check catches incorrect or misspelled localization keys.

GUI Effect Checker One of the most prevalent GUI-related bugs is *invalid UI update* or *invalid thread access*: accessing the UI directly from a background thread.

Most GUI frameworks (including Android, AWT, Swing, and SWT) create a single distinguished thread — the UI event thread — that handles all GUI events and updates. To keep the interface responsive, any expensive computation should be offloaded to *background threads* (also called *worker threads*). If a background thread accesses a UI element such as a JPanel (by calling a JPanel method or reading/writing a field of JPanel), the GUI framework raises an exception that terminates the program. To fix the bug, the background thread should send a request to the UI thread to perform the access on its behalf.

It is difficult for a programmer to remember which methods may be called on which thread(s). The GUI Effect Checker [32] solves this problem. The programmer annotates each method to indicate whether:

- It accesses no UI elements (and may run on any thread); such a method is said to have the "safe effect".
- It may access UI elements (and must run on the UI thread); such a method is said to have the "UI effect".

The GUI Effect Checker verifies these effects and statically enforces that UI methods are only called from the correct thread. A method with the safe effect is prohibited from calling a method with the UI effect.

Signedness Checker The Signedness Checker guarantees that signed and unsigned values are not mixed together in a computation. In addition, it prohibits meaningless

operations, such as division of an unsigned value. The Signedness Checker uses type annotations to indicate the signedness that the programmer intends an expression to have.

Signedness is primarily about how the bits of the representation are interpreted, not about the values that it can represent. An unsigned value is always positive, but just because a variable's value is positive does not mean that it should be marked as `@Unsigned`. If variable *v* will be compared to a signed value, or used in arithmetic operations with a signed value, then *v* should have signed type.

Constant Value Checker The Constant Value Checker is a constant propagation analysis: for each variable, it determines whether that variable's value can be known at compile time. Whenever all the operands of an expression are compile-time constants (that is, their types have constant-value type annotations), the Constant Value Checker attempts to execute the expression. This is independent of any optimizations performed by the compiler and does not affect the code that is generated.

The Constant Value Checker statically executes operators that do not throw exceptions (e.g., `+`, `-`, `<<`, `!=`), and also calls to methods annotated with `@StaticallyExecutable`.

Aliasing Checker The Aliasing Checker identifies expressions that definitely have no aliases.

Two expressions are aliased when they have the same non-primitive value; that is, they are references to the identical Java object in the heap. Another way of saying this is that two expressions, *exprA* and *exprB*, are aliases of each other when *exprA*==*exprB* at the same program point.

Assigning to a variable or field typically creates an alias. For example, after the statement `a = b;`, the variables `a` and `b` are aliased.

Knowing that an expression is not aliased permits more accurate reasoning about how side effects modify the expression's value.

4.10 OTHER IMPROVEMENTS

Compound checkers Sometimes, it is desirable to run multiple checkers in tandem; running one checker as a pre-pass can improve the precision of a downstream type system. For example, the Nullness Checker uses the Map Key Checker to decide if the result of a `map.get` call can be non-null. To leverage existing checkers in this way, we designed a way to run two (or more) checkers. A *compound checker* is one that uses other checkers. A compound checker can access the annotated type computed by a subchecker. The Constant Value Checker is often used by other checkers in this manner.

Eclipse plug-in We created an Eclipse plug-in that improves integration of our type systems with the Eclipse IDE. This was in response to requests from people who were annotating apps. The plug-in integrates errors into the Eclipse Problem View and allows customization of how the type-checker is run and which type-checkers are run.

Support for Java 8 features All our type systems now correctly check subtyping relationships involving lambdas, method references, and default methods of interfaces — these are language features new to Java 8. We also made a number of other changes,

including adapting to changes in the Java 8 specification: annotation syntax for receivers (the "this" parameter), default behavior of the `@Target` meta-annotation, etc.

Mutation and polymorphism We discovered that when a type system uses all three of type variables, mutation, and polymorphic qualifiers, then it is unsound. This helps to explain seemingly-odd design choices by previous languages, and it prevents users from creating unsound type systems.

Qualifier Framework The Checker Framework represented a type qualifier as a Java annotation. However, it can be useful to give type qualifiers richer data and behavior. Therefore, we designed a variant of the Checker Framework in which a user-defined object can act as a type qualifier. We implemented a simple version of the information-flow checker using the qualifier framework. The implementation supports qualifier polymorphism that is sound even when it is used in combination with annotated type variables and mutation. It does not support the more advanced features of the information-flow checker.

5 CONCLUSIONS

We have described a collaborate verification model for high assurance app stores, in which app developers provide annotated source code whose information flow properties are verified by the app store's auditors. In this model, the application developer and the auditor each do tasks that are easy for them, reducing the overall cost.

We designed IFT, a flow-sensitive, context-sensitive type system that enables collaborative verification of information flow properties in Android applications. Its design focuses on usability and practicality, and it supports a rich programming model.

We evaluated IFT by analyzing 72 new applications (57 of them malicious), which were written by 5 different corporate Red Teams who were not under our control. IFT detected 96% of the information flow-related malware (Sect. 3.2.9 describes an extension to IFT that would increase this to 100%) and 82% of all malware. Other experiments show that IFT is easy to use for both programmers and auditors, making a collaborative verification model practical for a high-assurance app store.

We have presented novel analyses for two programming paradigms — Java reflection and Android intents — that are useful to programmers but challenging for static analysis. Our analyses statically resolve reflection targets and intent payloads. Though sound and conservative, they achieve high precision in practice, as confirmed by experiments on real-world Android apps. Our implementations are publicly available as open source, and they can be integrated with an arbitrary downstream analysis to improve its precision.

Our system is freely available at `http://types.cs.washington.edu/sparta/`, including source code, library API annotations, user manual, and example annotated applications.

6 REFERENCES

[1] Y. Agarwal and M. Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *MobiSys*, pages 97–110, 2013.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP 2013 — Object-Oriented Programming, 27th European Conference*, pages 378–400, Montpellier, France, July 3–5, 2013.

[3] Android NDK. http://developer.android.com/tools/sdk/ndk/index.html/.

[4] APKParser. https://code.google.com/p/xml-apk-parser/.

[5] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *IEEE S&P*, 2012.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269, 2014.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014, Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, pages 259–269, Edinburgh, UK, June 9-11, 2014.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 217–228, Raleigh, NC, USA, October 16–18, 2012.

[9] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE S&P*, 2008.

[10] J. Bloch. *Effective Java Programming Language Guide*. Addison Wesley, Boston, MA, 2001.

[11] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Waikiki, Hawaii, USA, May 25–27, 2011.

[12] C. Bonnington. First instance of iOS app store malware detected, removed, 2012. http: //www.wired.com/gadgetlab/2012/07/first-ios-malware-found/.

[13] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Boston, MA, USA, 1975.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, Bethesda, MD, USA, June 29–July 1, 2011.

[15] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *16th USENIX Security Symposium*, Boston, MA, USA, August 8–10, 2007.

[16] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–513, 1977.

[18] W. Dietl, S. Dietzel, M. D. Ernst, K. Mus¸lu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[19] M. Egele, C. Kruegel, E. Kirdaz, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, 2011.

[20] M. Egele, C. Kruegel, E. Kirdaz, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *18th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 7–9, 2011.

[21] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid:

an information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX 9th Symposium on OS Design and Implementation*, Vancouver, BC, Canada, October 4–6, 2010.

[22] K. Engelhardt, R. van der Meyden, and C. Zhang. Intransitive noninterference in nondeterministic systems. In *CCS*, 2012.

[23] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM Conference on Computer and*

Communications Security (CCS), pages 1092–1104, Scottsdale, AZ, USA, November 4–6, 2014.

[24] M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine. Locking discipline inference and checking. In *ICSE'16, Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, USA, May 18–20, 2016.

[25] M. D. Ernst, D. Macedonio, M. Merro, and F. Spoto. Semantics for locking specifications. In *8th NASA Formal Methods Symposium*, Minneapolis, MN, USA, June 7–9, 2016.

[26] F-Droid. Free and open source Android app repository. http://f-droid.org, Feb 2014.

[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 627–638, Chicago, IL, USA, October 18–20, 2011.

[28] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *SPSM*, 2011.

[29] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *1997 IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 4–7, 1997.

[30] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, November 2009.

[31] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, pages 291–307, Vienna, Austria, June 13–15, 2012.

[32] C. S. Gordon, W. Dietl, M. D. Ernst, and D. Grossman. JavaUI: Effects for controlling UI object access. In *ECOOP 2013 — Object-Oriented Programming, 27th European Conference*, pages 179–204, Montpellier, France, July 3–5, 2013.

[33] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, pages 1025–1035, 2014.

[34] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *18th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 6–8, 2012.

[35] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294, Low Wood Bay, UK, June 26–28, 2012.

[36] A. Greenberg. iPhone security bug lets innocent-looking apps go bad. http:

//www.forbes.com/sites/andygreenberg/2011/11/07/iphone-security-bug-letsinnocent-looking-apps-go-bad/, 2011.

[37] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE S&P*, 2011.

[38] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, pages 87–96, Arlington, VA, USA, March 13–15, 2006.

[39] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 639–652, Chicago, IL, USA, October 18–20, 2011.

[40] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the Thirteenth International World Wide Web Conference*, pages 40–52, New York, NY, USA, May 17–20, 2004.

[41] M. Isaac. Android malware found in angry birds add-on apps. http://www.wired.com/2011/ 06/android-malware-angry-birds/, 2011.

[42] C. Jones. *The Economics of Software Quality*. Addison-Wesley, 2011.

[43] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[44] M. Kassner. Google Play: Android's Bouncer can be pwned. http://www.techrepublic.com/ blog/it-security/-google-play-androids-bouncer-can-be-pwned/, 2012.

[45] C. Kitching and L. McVoy. BK2CVS problem. http://lkml.indiana.edu/hypermail/ linux/kernel/0311.0/0635.html, 2003.

[46] D. Kravets. Android market apps hit with malware. http://www.wired.com/2011/03/ android-malware-2/, 2011.

[47] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *Compiler Construction: 14th International Conference, CC 2005*, pages 287–304, Edinburgh, Scotland, April 2005.

[48] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, 2013.

[49] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. Technical Report CS-13-02, Brown University, 2013.

[50] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden,

D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE'15, Proceedings of the 37th International Conference on Software Engineering*, Florance, Italy, May 20–22, 2015.

[51] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop*, pages 16–27, Venice, Italy, July 5–6, 2006.

[52] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP 2014 — Object-Oriented Programming, 28th European Conference*, pages 27–53, Uppsala, Sweden, July 30–August 1, 2014.

[53] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *18th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 6–8, 2012.

[54] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Third Asian Symposium on Programming Languages and Systems*, pages 139–160, Tsukuba, Japan, November 2005.

[55] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *Proceedings of the 2012 ACM Symposium on Applied Computing*, pages 1457–1462, Trento, Italy, March 27–30, 2012.

[56] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Fifteenth International Symposium on Software Reliability Engineering*, pages 198–209, Saint-Malo, France, November 3–5, 2004.

[57] S. McConnell. Software Estimation: Demystifying the Black Art. Microsoft Press, 2006.

[58] A. Milanova and W. Huang. Composing polymorphic information flow systems with reference immutability. In *FTfJP 2013: 14th Workshop on Formal Techniques for Java-like Programs*, pages 5:1–5:7, Montpellier, France, July 1, 2013.

[59] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL '99, Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 20–22, 1999.

[60] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java + information flow. http://www.cs.cornell.edu/jif.

[61] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS*, 2006.

[62] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *FSE 2012, Proceedings of the ACM SIGSOFT 20th Symposium on the Foundations of Software Engineering*, pages 6:1–6:11, Cary, NC, USA, November 13–15, 2012.

[63] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE'15, Proceedings of the 37th International Conference on Software Engineering*, Florance, Italy, May 20–22, 2015.

[64] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium*, pages 543–558, Washington, DC, USA, August 14–16, 2013.

[65] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference*, pages 340–349, December 9–11,, Honolulu, HI, USA 2009.

[66] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security*, pages 527–542, 2013.

[67] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[68] N. J. Peroco and S. Schulte. Adventures in BouncerLand. In *Black Hat USA*, Las Vegas, NV, USA, July 25–26, 2012.

[69] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL 2002, Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–330, Portland, Oregon, January 16–18, 2002.

[70] F. Rashid.Android malware makes up this week's dangerous apps list. https:

//www.appthority.com/news/android-malware-makes-up-this-weeks-dangerousapps-list, 2013.

[71] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[72] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, September 2003.

[73] R. Schouwenberg. Malware in the amazon app store. https://www.securelist.com/en/blog/208194054/Malware_in_the_Amazon_App_Store, 2012.

[74] B. C. Smith. Procedural reflection in programming languages. Technical Report MIT-LCS-TR272, MIT Laboratory for Computer Science, Cambridge, MA, January 1982.

[75] S. Smith and M. Thober. Refactoring programs to secure information flows. In *PLAS 2006: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 75–84, Ottawa, Canada, June 10, 2006.

[76] P. Su. Broken Windows theory. http://blogs.msdn.com/b/philipsu/archive/2006/06/14/631438.aspx, 2006.

[77] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *SAS*, 2004.

[78] M. Tatsubori. Living with reflection: Towards coexistence of program transformation by middleware and reflection in Java applications. In *6th JSSST Workshop on Programming and Programming Languages (PPL2004)*, Gamagohri, Aichi, Japan, March 11–13, 2004.

[79] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the Twelfth International Symposium on Static Analysis, SAS 2005*, pages 352–367, London, UK, September 7–9, 2005.

[80] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *ObjectOriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 281–293, Minneapolis, MN, USA, October 15–19, 2000.

[81] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *ObjectOriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[82] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Web 2.0 Security and Privacy Workshop*, Oakland, CA, USA, May 26, 2011.

[83] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[84] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 607–621, Lille, France, April 14–18, 1997.

[85] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When benign apps become evil. In *22nd USENIX Security Symposium*, pages 559–572, Washington, DC, USA, August 14–16, 2013.

[86] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst. A format string checker for Java. In *ISSTA* 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 441–444, San Jose, CA, USA, July 23–25, 2014.

[87] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst. A type system for format strings. In *ISSTA* 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 127–137, San Jose, CA, USA, July 23–25, 2014.

[88] X. Xiao, N. Tillmann, M. Fähndrich, J. De Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In *ASE 2012: Proceedings of the 27th Annual International Conference on Automated Software Engineering*, pages 80–89, Essen, Germany, September 5–7, 2012.

[89] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *21st USENIX Security Symposium*, Bellevue, WA, USA, August 8–10, 2012.

[90] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *21st USENIX Security Symposium*, Bellevue, WA, USA, August 8–10, 2012.

[91] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA 2012, Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 243–253, Minneapolis, MN, USA, July 17–19, 2012.

[92] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *2003 IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, CA, USA, May 12–14, 2003.

[93] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE S&P*, 2012.

[94] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *18th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 6–8, 2012.